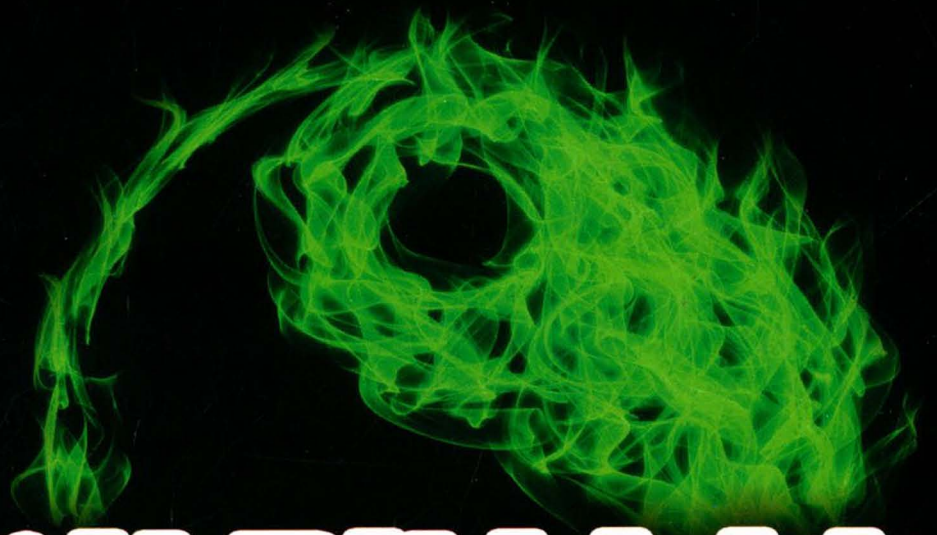
Broadview
www.broadview.com.cn

严格限于指导从业者查防外挂　坚决反对任何恶意或不法行为



游戏外挂攻防艺术

徐胜　著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

游戏外挂攻防艺术

徐胜 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

随着网络的普及，网络游戏得到了众多网民的青睐。但是，网络游戏的盛行，也给游戏玩家和游戏公司带来了很多安全问题，如木马盗号、外挂作弊等。对于正常的游戏玩家和游戏公司来说，外挂的危害尤其突出。因为一款免费的外挂，不仅可能携带游戏木马，还会影响游戏的平衡，甚至伤害其他玩家的感情。虽然很多游戏玩家和安全爱好者对外挂和反外挂技术有强烈的兴趣，但目前市面上很难找到一本能够深入浅出地讲解这部分知识的书。本书将带领读者走近外挂和反外挂技术这个神秘的领域，让读者了解外挂的制作过程、作弊过程以及反外挂检测技术，从而提升读者对游戏安全的认识。

本书是作者长期分析外挂软件和反外挂的经验所得，分 5 篇，共 10 章，包括游戏和外挂初识、外挂技术、游戏保护方案探索、射击游戏安全和外挂检测技术。本书内容循序渐进，层层解剖外挂涉及的一些关键技术，包括注入、隐藏、交互、Hook 和 Call 函数等，让读者对外挂产生直观和深刻的认识，独创性的外挂分析和检测方法对安全从业者而言也有很好的借鉴意义。

图书在版编目（CIP）数据

游戏外挂攻防艺术 / 徐胜著. —北京：电子工业出版社，2013.2
ISBN 978-7-121-19532-7

Ⅰ. ①游… Ⅱ. ①徐… Ⅲ. ①互联网络—游戏—安全技术 Ⅳ. ①G899 ②TP393.08

中国版本图书馆 CIP 数据核字（2013）第 020169 号

策划编辑：张春雨
责任编辑：张春雨
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787 × 980　1/16　　印张：16　　字数：420 千字
印　　次：2013 年 2 月第 1 次印刷
印　　数：4000 册　　定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

序

网络游戏产业是一个新兴的朝阳产业。经历了 20 世纪末至今的快速发展，我国的网络游戏产业正处于高速成长并快速走向成熟期的阶段。目前，网络游戏产业已成为我国网络经济及文化娱乐产业的重要支撑，因此，我国的网络游戏产业还需要国家和政府的鼓励、支持与呵护——尤其是在政策及法律法规层面。与此同时，也更需要学术界和产业界重视网络游戏的安全威胁，积极研究网络游戏的安全技术，为我国的网络游戏产业创造一个健康、良性的发展环境。

当前，网络游戏面临的头号安全威胁就是网络游戏外挂。近些年来，信息安全技术研究领域的著作很多，但专门针对网络游戏安全研究的可谓凤毛麟角，而且基本上是以网络游戏外挂现象的揭示和平铺直述的文字为主，真正揭开现象背后的本质、讨论反外挂技术的著作还没有看到。《游戏外挂攻防艺术》一书填补了这方面图书的空白，首先揭示了网络游戏及其外挂的原理，然后通过代码实例罗列和盘点了网络游戏外挂的注入技术、无模块化隐藏技术、交互通信隐藏技术、函数调用隐藏技术、Hook 技术以及游戏的安全保护，最后抛砖引玉，提出了外挂检测与防御的技术。

网络游戏产业中，网络游戏外挂在巨大的灰色利益的驱动下，不会因为我们对它的厌恶而消亡，相反，它会“道高一尺，魔高一丈”。学术界和产业届的学者和技术专家们只有正视它的存在，深刻地了解和认识它，并在技术上积极主动地对抗它，才能真正保护网络游戏和网络游戏产业。

谨将此书推荐给热衷和从事信息安全和网络游戏安全工作的专业技术人员学习和参考。

电子科技大学　教授
李毅超

前　言

我的安全之路

2002 年，在父亲的陪伴下，我登上了西去的火车，来到了“三国”发祥地四川南充就读大学。在大学期间，虽然我很用心地学习与计算机相关的每门功课，但是因为信息闭塞，对计算机这门科学，我始终觉得自己没有入门，因而深感苦闷。

直到 2003 年，得知网易创始人丁磊成为中国首富之后，我才知道成都还有一所大学叫做电子科技大学。2006 年，我很幸运地进入电子科技大学网络攻防实验室就读研究生，从此正式开始了网络安全的学习和研究之旅。

2007 年，我们实验室接到第三方公司的一个主动防御系统项目，我正好负责实现其中的防火墙模块。这是我第一次独立负责设计和开发一个完整的安全项目。经过这个项目的洗礼，我对 Windows 操作系统、木马和 Rootkit 技术有了比较深入的认识——最大的收获还是学会了如何去学习和做技术研究。

2009 年，从电子科技大学毕业之后，我坐上了南下的火车，来到深圳一家互联网公司从事反外挂分析工作。一年之后，由于种种原因，我毅然选择离开。

2010 年，我再次回到成都，在中兴通讯成都研究所开始了手机软件开发之旅。不过，当时我能接触的只是低端的特性机，不是一直渴望的智能机，所以，3 个月之后，我再次选择离开。

2011 年，我来到淘宝做无线安全，终于开始接近智能机。无线安全是一个新的方向，包括 WAP 安全、Android 安全、iOS 安全等，涉及的安全风险点比较多，资料也比较匮乏，但是，我们依然贡献了自动化探测 Android 软件漏洞的 fuzzing 系统、基于 Smali Hook 技术的 Android 风险分析系统和 iOS 静态扫描工具等。

2012 年对我来说是非常不平凡的一年，因为我完成了这本书，在安全研究的道路上留下了自己的足迹。虽然不一定能做到字字珠玑，但我相信，这本书里体现的各种思想，包括差异分析、模糊假定、虚假注册等，还是会引起安全从业者的共鸣。当然，本书出版时，我可能已经退出接触多年的安全行业，转战无线产品的研发。

我相信，无论自己在做什么，内容不是最重要的，关键是能不能把事情做到极致，能不能以创新的思维去解决问题。

我相信，无论自己是否走在安全研究的道路上，方向、专注、坚持和自信都是打开成功之门的必备钥匙。

写作目的

为什么要写这本书？

反外挂是一项系统工程，涉及多方面的知识，包括对游戏的理解、对 Windows 操作系统的认识、对网络协议的掌握、对调试与反调的掌握等，需要具备技术创造性才能在攻防中占据优势。但是，目前市面上还没有一本深入讲解这部分知识的书。所以，如果有一本书能深刻剖析游戏、外挂和反外挂，将是对网络游戏安全感兴趣的读者的福音——因为它能够提升读者对网络游戏安全的认识、开阔眼界。

关于本书

在计算机领域，无论是网络、操作系统、脚本，还是软件，都有各种层次的书籍来描述对应的安全知识，但是对网络游戏而言，国内讲解其安全性的书籍却少之又少。国外有一本书叫做《网络游戏安全揭秘》（*Exploiting Online Games: Cheating Massively Distributed Systems*），其对外挂技术的剖析只是蜻蜓点水，不够深入，对游戏架构等也未做详细描述，而且未对反外挂思路进行整理，所以，对一个急于了解

外挂和游戏原理的人来说，只能是望梅止渴。为了让读者了解外挂的制作过程和原理，理解游戏的核心概念，理清反外挂的思路，笔者花费了大量时间投入本书的写作，包括实例的编写、各种原理图的绘制等。

本书有很多亮点，相信值得读者好好花精力去研读。这些亮点包括但不限于以下方面。

- 与同类技术作对比，并进行优缺点的评价。
- 不仅剖析外挂技术，也罗列反外挂技术。
- 解决问题的思路新颖，如差异分析、模糊假定、线程转移、消息分流等。

实例程序及相关文档

本书的每一章都提供了实例程序及相关阅读文档，放在与各章对应的资源包中。读者可以访问 http://www.broadview.com.cn/19532 下载，或者加入 QQ 群 143914331 获取。

读者在阅读本书以及相关实例程序和文档的过程中遇到的困惑，都可以通过给 gamebot@foxmail.com 发送电子邮件的方式来获取满意的答案。

免责声明

本书所讨论的技术仅用于研究和学习，旨在提高游戏的安全性，严禁用于不良动机。任何个人、团队、组织不得使用其进行非法活动，否则后果自负。

本书结构

本书分为 5 篇，共 10 章。

第 1 篇是“游戏和外挂初识篇”。本篇是全书的开篇，主要从内存对象的角度向读者阐明什么是外挂，外挂带来的危害，以及应该怎样理解外挂与游戏的关系。

第 2 篇是“外挂技术篇”。本篇包括第 2 章至第 7 章，共 6 章，主要阐述开发一款优秀的外挂所采用的技术。这些技术分别是注入技术、模块隐藏技术、安全的交

互技术、Call 函数技术、Hook 技术和自我保护技术。本篇的每一章都包括同类技术的优缺点分析和大量的实例程序，相信根据这种安排，读者能够对外挂的制作过程和作弊过程有深刻的认识，同时，每一章中对同类技术的对比，也会帮助读者理解和掌握相应的技术。

第 3 篇是“游戏保护方案探索篇”。本篇主要带领读者分析游戏的保护方案，帮助读者掌握通用的分析方法。虽然现实环境中具体的保护方案可能千差万别，但是通过本篇的学习，读者能够从宏观上掌握分析的目标和方法，从而在新的保护方案面前有的放矢。

第 4 篇是“射击游戏安全专题”。本篇主要围绕第一人称射击类游戏展开。因为此类游戏的安全问题比较相似，所以本书专门用一篇来阐述此类游戏可能涉及的一些安全风险，以便读者集中了解这方面的安全问题。

第 5 篇是“外挂检测技术篇”。本篇主要向读者展示一些比较实用的外挂检测技术，其中不乏新颖的检测方案，希望能起到抛砖引玉的作用。

致谢

感谢我的父母，你们给了我一个能历经风雨的好身体。特别感谢我的父亲，是你坚持相信我能行，才又再次印证那句古语——浪子回头金不换。

感谢我的妻子，这么多年跟我东奔西走，陪在我身边，即使天各一方，依然默默地支持我，嗷嗷待哺的儿子也是你在带，感谢之辞已经无法承载这份情谊。

感谢我的好友唐仙、杨海燕、陈殊聪，是你们在我考研那段时间给予的鼓励和帮助让我选择了坚持，使我最终以平和的心态取得好成绩，愿友谊天长地久。

感谢电子科技大学网络攻防实验室的李毅超老师、刘丹老师和曹越老师，特别感谢李毅超老师抽出宝贵时间对书稿提出建议并作序。感谢黄沾、何子昂、梁晓等师兄和师姐的帮助，感谢杨宇、阳广元、钱彦江、舒伯承、覃丽芳等同门的鼓励，感谢罗尧、康凯、申文迪、刘泽鹏等师弟的支持，怀念在电子科技大学与你们并肩作战的日子。

感谢阿里巴巴安全技术团队的张玉东、中国科学院的张谦博士以及成都安思科技公司的沈东、任云韬对本书的宝贵建议和推荐。

感谢博文视点的张春雨以及他的团队，是你们的努力使本书最终能与广大读者见面。你们的专业意见给了我很多帮助，也开阔了我的视野。你们的效率和敬业精神给我留下了深刻的印象，让我相信国内技术类图书会有美好的未来。

最后特别感谢历史长河中那些曾在逆境中成才的先哲们，你们是人类文明的瑰宝，是所有处于逆境的有志之士的楷模。

联系方式

新浪微博：http://weibo.cn/winsunfish83

腾讯微博：http://t.qq.com/winsun_xu_GG

博　　客：http://blog.csdn.net/winsunxu

电子邮箱：gamebot@foxmail.com

目　　录

第 1 篇　游戏和外挂初识篇

第 2 篇　外挂技术篇

第 3 篇　游戏保护方案探索篇

第 4 篇　射击游戏安全专题

第 5 篇 外挂检测技术篇

第 1 篇

游戏和外挂初识篇

第 1 章　认识游戏和外挂

网络游戏和外挂一直是游戏爱好者们关注的对象，但对于大多数玩家来说，目前仅限于简单的开外挂来辅助游戏。然而，对于计算机技术爱好者们来说，他们更希望深入了解游戏外挂技术和反外挂技术。

本章将带领读者从宏观角度，从 4 个主要方面理清游戏和外挂涉及的核心概念：内存挂与游戏的关系；游戏的资源、协议和内存对象布局；外挂的设计思路；反外挂的思路。

1.1　游戏安全现状

网络游戏是一种特殊的商品。随着时间的推移和新玩家的进入，网络游戏中的装备、道具等虚拟财产开始具有真实的价值，可以用现金进行交换。从此，网络游戏面临严重的安全问题。

360 安全中心发布的《2011 年上半年中国网络游戏产业安全报告》显示，在遭受“网络窃贼”攻击的游戏玩家中，有三分之二的玩家虚拟装备被盗，有三分之一的玩家游戏账号被盗，安全问题已经成为玩家和厂商、玩家和玩家之间发生纠纷的主要因素。

近年来，随着网络游戏大行其道，网络游戏的安全对抗也变得日益尖锐。除了来自盗取游戏账号和装备的木马的威胁，外挂成为游戏厂商最头痛的安全问题之一。

目前，市面上的外挂辅助软件形形色色，无奇不有。不仅大量的外挂程序对游戏的公平性造成了极大的破坏，而且各种免费外挂常常内藏“玄机”，包含游戏木马。所以，一旦玩家使用了一款看似免费的外挂软件，就要为此付出游戏账号被盗的风险成本。

同时，外挂对游戏厂商来说是一种具有双重危害的程序。因为各款游戏在玩法、故事情节、通信协议和设计架构等方面没有统一的标准，所以游戏的安全机制必须独立设计，这无疑增加了大量的运营和维护成本。对于很多小型游戏公司来说，这是一笔无法承受的开支。

下面就让我们了解一下外挂和游戏的关系。

1.2　什么是外挂

在很多游戏玩家看来，外挂是一种很神奇的东西。其实，外挂就是一种特定的辅助软件。外挂虽然被网络游戏公司所痛斥，并视为恶意软件，但它还是有别于木马程序，因为，外挂是用户主动使用的，而木马则是在用户的终端上偷偷运行的。

我们可以根据外挂是否有模块进入游戏客户端，把外挂大致分成两大类，即内存挂和非内存挂。

内存挂是指会释放核心功能模块，并将其注入游戏客户端的外挂。内存挂的核心功能模块会对游戏客户端实施各种侵入性操作，一般有修改游戏代码或数据、Hook 游戏代码、Call 游戏函数等行为。

非内存挂无须注入模块到游戏进程空间。非内存挂因为切入点的不同而各有千秋，如修改游戏资源脚本的资源挂、模拟游戏客户端收发包的协议挂、模拟用户按键操作的按键挂等。

1.3　内存挂与游戏的关系

在这一节中，我们将主要讨论内存挂与游戏的关系。对非内存挂，我们会在第 1.4 节讲解。

对内存挂而言，如果从模块的角度看，它主要由一个负责启动的 EXE 程序和一个或若干个实现核心功能的 DLL 或 SYS 模块组成。从更为形象的角度，我们可以把

外挂看作一把装了子弹的枪——枪本身是负责启动的 EXE 程序，而子弹是实现核心功能的模块，当某个特定条件发生（如目标游戏进程启动等），子弹就会射出，进而击中目标。内存挂的工作目的是释放核心模块并将其注入目标游戏进程。

图 1–1 从内存布局结构上描绘了内存挂。

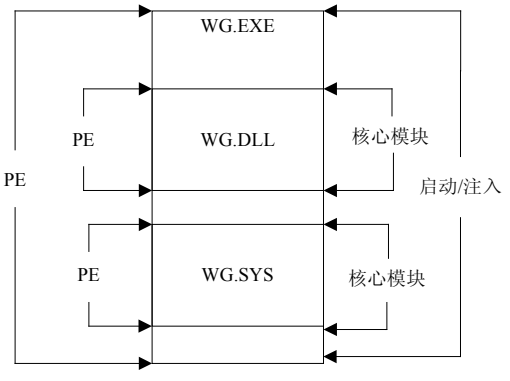


图 1-1　内存挂的内存布局

从图 1–1 中可知，负责启动/注入的 EXE 模块与核心模块之间是包含与被包含关系。针对这种 PE 打包方式，有以下两种情况。

- 核心模块以资源的形式打包在外挂的启动 EXE 模块中。
- 核心模块以数据的形式打包在外挂的启动 EXE 模块中。

内存挂在启动之后，首先会遍历进程以定位目标游戏进程，然后将核心功能模块（如 WG.DLL）释放到驱动目录下，把 WG.DLL 注入游戏进程空间，同时退出自己的启动模块，以防止被反外挂系统检测到。之后，外挂的作弊功能都由核心模块负责提供。

内存挂与游戏的内存布局如图 1–2 所示，我们可以从中看出内存挂与游戏内存布局的关系。就像微生物世界里的寄生虫与宿主之间的关系一样，寄生虫的生存空间和时间依赖于宿主，同样的道理，外挂的生成空间和时间也依赖于特定游戏的特定版本。因为每当游戏更新版本之后，之前外挂获取的关键数据点、Call 函数的地址等往往会发生变化，所以外挂也要更新，发布新版本——反外挂和外挂不停地在攻防对抗中锻炼着彼此的思维能力和耐力。

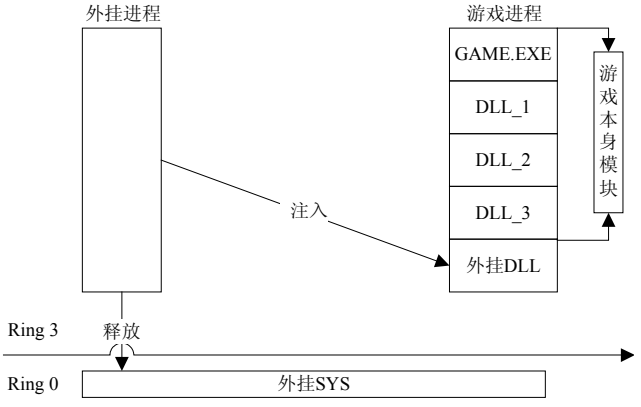


图 1-2　内存挂与游戏的关系

外挂 DLL 包含了内存挂所拥有的全部核心功能（假设该外挂无驱动），包括自我隐藏、Call 函数、修改代码或数据、Hook 等。对这些细节，本书的后续章节中会结合外挂的常用功能（如加速、无敌、吸怪、自动攻击等）进行详细剖析，并从反外挂的角度给出防御方案，让读者在外挂与反外挂的思维对抗中学到思考方法和技术。

1.4　游戏的 3 个核心概念

开发外挂，不仅需要良好的逆向分析和调试能力，还需要对游戏有一定的了解。正所谓有的放矢——有了对游戏的认识这一基础，就能更快地分析并找出目标。本节就带领读者简单地认识游戏。

从本质上看，游戏是由数据和代码构成的。数据包括资源、协议、内存对象和按键等；而代码则构成了游戏逻辑。这里的每个方面都涉及很多知识，都可以单独写成一本书。所以，本书会从外挂开发者最感兴趣的游戏资源、协议和内存对象这 3 个方面来阐述游戏。

1.4.1　游戏资源的加/解密

游戏资源是指为游戏引擎提供的图片、声音、地图、动画等原始素材文件。早期游戏中的这些原始素材文件，可能只是在游戏客户端分门别类地散落在游戏安装

目录下，但是，现在的游戏客户端为了提高读取资源文件的 I/O 效率和保护游戏资源不被破解，常把这些原始素材文件打包和加密，使游戏客户端安装目录下只剩一个类似 script.pvf 或 script.dat 形式的资源文件。资源文件的加/解密理论知识，读者可以参考《揭秘数据解密的关键技术》一书。

如图 1–3 所示：本章的资源包中有一个用笔者设计的打包器打包的资源文件 winsun.pvf，这个资源文件包含若干个目录和若干个文件，每个目录下有一个文件；还有一个 DecrptResource.exe 程序，这个程序会根据资源文件 progressvalue.dat 中的值来改变进度条的移动速度。

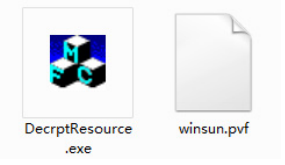


图 1-3　解密脚本文件的程序和资源文件

启动 DecrptResource.exe 程序，显示界面如图 1–4 所示。单击“开始”按钮，进度条会在向前移动 500 000 步之后结束运行，如图 1–5 所示。

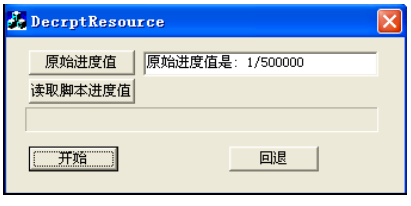


图 1-4　DecrptResource.exe 运行界面

图 1-5　以 1/500000 的速度移动进度条

如果单击“回退”按钮，进度条将退回如图 1–4 所示的样子。如果单击“读取脚本进度值”按钮，进度值会发生改变，同时会解密资源文件 winsun.pvf 并释放整个资源目录包中的文件，如图 1–6 和图 1–7 所示。

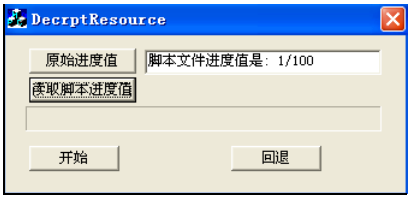


图 1-6　读取脚本进度值

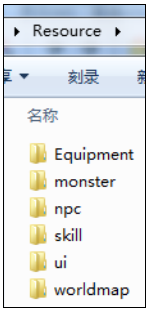


图 1-7　winsun.pvf 释放的脚本目录和文件

这个时候，单击“开始”按钮，进度条将只前移 100 步就结束运行。

上面这个例子模拟的是游戏的解密资源文件和获取资源文件关键数据的过程。当然，如图 1–7 所示的过程只是为了展示方便，现实中的游戏往往不会把整个目录和文件释放出来暴露在客户端。

下面，我们从静态和动态的角度分析一下 winsun.pvf 这个资源文件，看看它是如何定位到解密入口的。在这个过程中，相信读者会明白游戏资源是怎么一回事，以及如何分析游戏资源。这个例子所涉及的打包器代码和解包器代码就不提供给读者了，希望读者能自己动手来破解 DecrptResource.exe 程序，从而彻底掌握资源文件 winsun.pvf 的打包和解包过程。

经过打包的资源文件，一般存储在游戏的安装目录下，而且体积都比较大，所以很容易找到。把游戏程序拉到 IDA 中进行反汇编，按下【Shift】+【F12】组合键，打开“Strings window”选项卡，看看是否有对这个资源文件的引用。

IDA 对 DecrptResource.exe 的反汇编结果如图 1–8 所示。

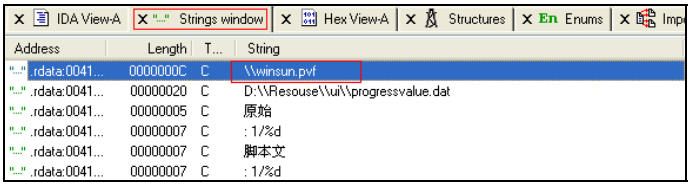


图 1-8　DecrptResource.exe 的 IDA Strings 窗口

在如图 1–8 所示的“Strings window”选项卡中，可以看到对资源文件 winsun.pvf 的引用。双击这个引用，来到如图 1–9 所示的界面。

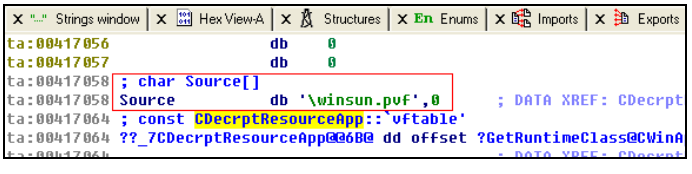


图 1-9　winsun.pvf 的字符串引用点

把鼠标放到字符串“Source[ ]”上并按下【X】键，定位到交叉引用点，如图 1–10 所示。

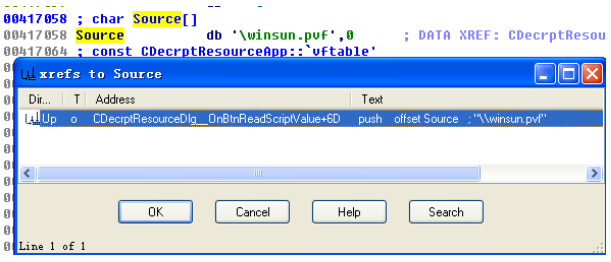


图 1-10 winsun.pvf 的代码引用点

如图 1-11 所示，字符串“\\winsun.pvf”的地址被压入栈顶，同时，下面还有一个“call j_ScriptDecrypt”的调用。很明显，资源文件就是 winsun.pvf，解密这个文件的程序就是 ScriptDecrypt 函数。只要结合动态调试工具，继续跟进 ScriptDecrypt 函数，就可以掌握整个解密过程。

```
.text:00401F36    cmp     esi, esp
.text:00401F38    call    __chkesp
.text:00401F3D    push    offset Source   ; "\\winsun.pvf"
.text:00401F42    lea     edx, [ebp+DstBuf]
.text:00401F48    push    edx             ; Dest
.text:00401F49    call    _strcat
.text:00401F4E    add     esp, 8
.text:00401F51    lea     eax, [ebp+DstBuf]
.text:00401F57    push    eax
.text:00401F58    call    j_ScriptDecrypt
```

图 1-11 跳到引用 winsun.pvf 的代码点

上面这些都是静态分析。游戏在使用资源文件之前，必须打开这个文件，而打开文件的动作，一般由 Windows 的 API CreateFile 或者 C 语言的库函数 fopen 等实现。所以，可以在 CreateFile 或 fopen 处设置下断点，通过动态调试的方法来定位解密的入口。

对于 ScriptDecrypt 函数实现过程的细节，就留给读者自己分析和研究了。这个过程需要耐心和细心，还需要一些想象和猜测。为了帮助读者更快地完成分析，下面给出资源文件 winsun.pvf 的设计图和部分数据结构的简要说明，希望能对读者破解 ScriptDecrypt 函数有所帮助。

图 1-12 winsun.pvf 的整体架构

winsun.pvf 资源文件由 3 个部分组成，如图 1-12 所示，分别是文件头、目录表和文件库。

文件头的数据结构如下。

```
// 文件头
typedef struct _FILE_HEAD{

    DWORD  dwFileNameLen;                // 文件名的长度
   BYTE    bFileName[dwFileNameLen];     // 文件名
    DWORD  dwFileDirSrcSize;             // 待解密的目录表大小
    DWORD  dwFileDirSrcKey;              // 解密文件目录的密钥
    DWORD  dwFileDirSum;                 // 文件目录结构的个数

}FILE_HEAD, *PFILE_HEAD;
```

目录表中目录项的结构体如下。

```
// 目录表中目录项的结构
typedef struct _FILE_DIR_TABLE_ENTRY{

    DWORD  dwDirLevel;              // 目录层数
    DWORD  dwDirFileNameLen;        // 带路径的文件名长度
    BYTE    bDirFileName[dwDirFileNameLen];     // 带路径的文件名
    DWORD  dwFileSize;              // 文件大小
    DWORD  dwFileDecryptKey;        // 解密该文件的密钥
    DWORD  dwFileOffset;            // 文件相对于文件库的偏移

} FILE_DIR_TABLE_ENTRY, *PFILE_DIR_TABLE_ENTRY;
```

目录表被解密之后，为了在内存中方便地定位目录表，程序会在内存中将 FILE_DIR_TABLE_ENTRY 结构重新映射成 MEM_DIR_TABLE_ENTRY。

```
// 内存文件目录表项，由 FILE_DIR_TABLE_ENTRY 映射过来
typedef struct _MEM_DIR_TABLE_ENTRY{

    DWORD  dwMemAddr;               // 结构体地址
    DWORD  dwStructLable;           // 结构体标识，便于区别结构的开始
    PWSTR  pwstrDirFileName;        // 指向新创建的空间，保存带路径的文件名
    DWORD  dwDirFileNameLen;        // 带路径的文件名长度
    DWORD  dwFileSize;              // 文件的大小
```

```
    DWORD  dwAllocFileBufLen;          // 实际要分配的存放文件的空间大小
        DWORD  dwFileDecryptKey;       // 解密该文件的密钥
        DWORD  dwFileOffset;           // 该文件相对于文件体的偏移
        PVOID    pNewFileBuf;

    // 如果是动画，就指向新创建的空间，以存放读出的文件内容
    }MEM_DIR_TABLE_ENTRY, *PMEM_DIR_TABLE_ENTRY;
```

接下来，将上面这些结构体与资源文件 winsun.pvf 对应，得出详细的整体架构，如图 1-13 所示，这是解密的结果。

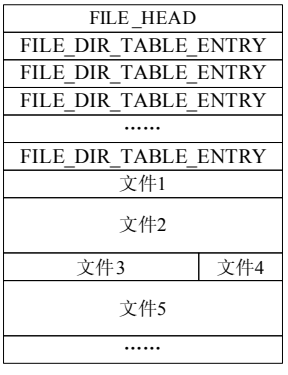


图 1-13　winsun.pvf 的详细架构

文件目录和文件库中的文件解密过程如图 1-14 所示。fread 函数根据文件头提供的目录信息读取目录到缓存中，然后调用 decrypt 函数解密缓存中的文件目录，得到 FILE_DIR_TABLE_ENTRY 结构体数组。

文件目录映射成内存目录的逻辑过程如图 1-15 所示。这个映射使游戏程序可以在内存中方便、快速地定位文件，而不用解密和释放整个目录。fread 函数根据 MEM_DIR_TABLE_ENTRY 提供的文件信息，读取文件库中对应的文件并将其放入 pBuff 中，然后调用 decrypt 函数来解密 pBuff 中的加密文件。

以上就是资源文件 winsun.pvf 的整个解密过程，希望能帮助读者认识资源的加密和打包。不过，在实际操作中，读者可能会碰到各种困难，需要具体问题具体分析。

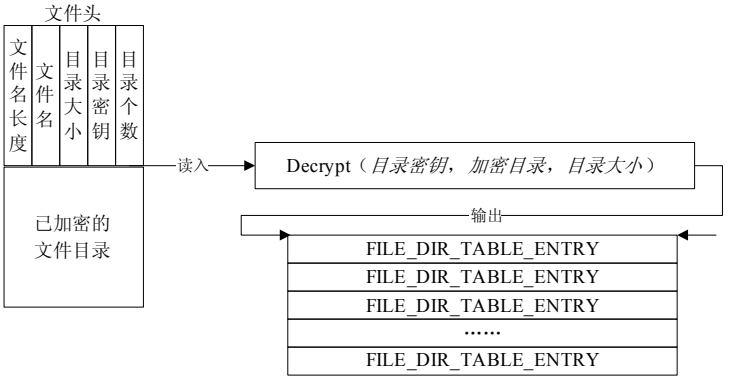


图 1-14　解密文件目录的过程

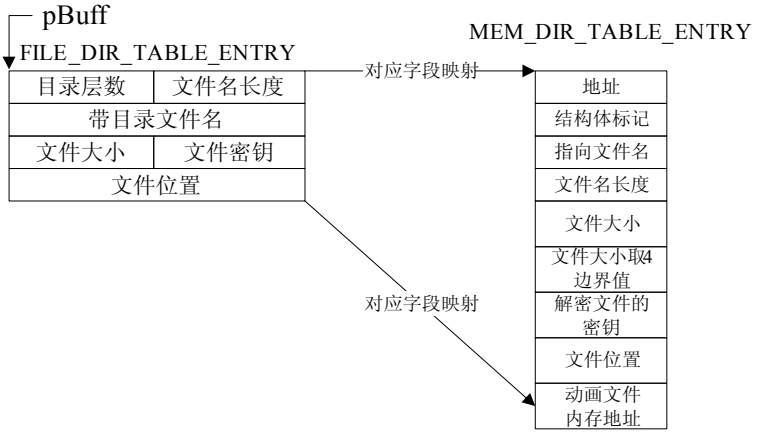


图 1-15　目录映射

1.4.2　游戏协议之发包模型

游戏协议是客户端与服务器以及客户端与客户端之间进行通信的基础。这里讨论的游戏协议是指 Windows 网络协议栈中的应用层协议。事实上，协议就是封装或解封数据所采用的某种特定格式的通信约定。当通信的一端按一定的格式封装数据并将其发送到另一端后，另一端再按同等格式来解封这些数据，这样就完成了一次通信。

在 Windows 网络协议栈中，游戏数据封装/解封的过程如图 1–16 所示。

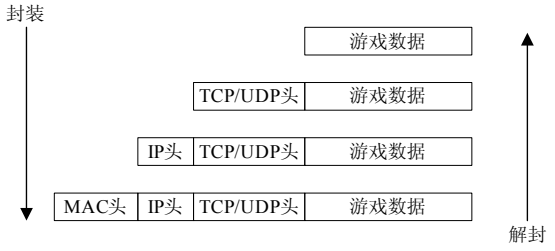


图 1-16　协议数据封装/解封的过程

图 1–16 展示的封装/解封过程都是由 Windows 系统中的协议模块处理的。游戏通信端为了快速分类处理游戏数据，也为游戏数据设计了相应的协议，即游戏采用某种格式来封装数据。

游戏中数据的格式划分各不相同，但是从宏观上看，通常采用数据包头加数据包体的格式。为了说明游戏协议数据的格式，下面先给出一种模拟格式。

```
// 数据包头
typedef struct _PACKET_HEAD
{
  BYTE   bPacketType;       // 包的类型，命令包为 1，通知包为 2
WORD  wProtocolId;          // 协议的 ID
  DWORD dwLength;           // 包的长度，包头加包体的字节数
DWORD dwCrc32;              // 包体的 CRC32 校验值，防止包体被篡改

}PACKET_HEAD, *PPACKET_HEAD;

// 数据包体
typedef struct _PACKET_BODY
{
DWORD dwPacketCount;        // 包的计数
BYTE   bBodyContent[1];     // 根据具体的游戏协议 ID 来确定内容
}PACKET_BODY, *PPACKET_BODY;

// 数据包
typedef struct _PACKET
{
   PACKET_HEAD stPacketHead;
```

```
    PACKET_BODY stPacketBody;
}PACKET,*PPACKET;
```

游戏客户端接收到用户的操作指令，然后根据具体的协议，把数据封装成包（Packet）发送出去。

在上面的模拟格式的基础上，为了帮助读者更好地理解游戏协议，下面给出一段用户摆摊出售物品的模拟行为代码。摆摊出售物品的行为至少涉及 3 条基本协议，分别是建立商店、摆上物品出售、关闭商店，对应的协议 ID 放在一个枚举类型中。

```
enum ENUM_PROTOCOL_ID
{
    ENUM_CMDPACKET_CREATE_SHOP,      // 建立商店
    ENUM_CMDPACKET_SELL_ITEMS,       // 摆上物品出售
    ENUM_CMDPACKET_CLOSE_ SHOP       // 关闭商店
};
```

下面我们看看将 ENUM_CMDPACKET_SELLER_ITEMS（摆上物品出售）这条协议封装到包中的过程。

```
// ENUM_CMDPACKET_SELLER_ITEMS 对应的数据内容
typedef struct _SELL_ITEMS{
DWORD  dwShopNameLen;                // 商店名字的长度
    CHAR    szName[dwShopNameLen];   // 商店的名字
    WORD  wItemColumNum;             // 物品在物品栏中的编号
    DWORD dwItemSellMoney;           // 某个物品的售价
    WORD  wShopColumNum;             // 商店栏编号
    DWORD dwSellCount;               // 同一个物品有多少个拿来出售
}SELL_ITEMS, *PSELL_ITEMS;

// 下面是伪代码
SELL_ITEMS stSellItem = {0};         // 初始化要出售商品的信息
stSellItem. dwShopNameLen = strlen("winsunshop");
// 初始化商店名称的长度
strncpy(stSellItem.szName, "winsunshop", strlen("winsunshop"));
// 初始化商店名称
```

```
stSellItem.wItemColumNum = 1;          // 物品栏编号
stSellItem.dwItemSellMoney = 1000;     // 物品的售价
stSellItem.wShopColumNum = 2;          // 商店栏编号
stSellItem.dwSellCount = 1;            // 出售个数
// 为数据包分配空间
DWORD dwPacketLen = sizeof(PACKET_HEAD) + 4 + sizeof(SELL_ITEMS);
PPACKET pPacket = new BYTE[dwPacketLen];
// 初始化包头
pPacket->stPacketHead.bPacketType = 0x01;          // 命令包
pPacket->stPacketHead. wProtocolId = ENUM_CMDPACKET_SELL_ITEMS;
// 摆摊出售商品
pPacket->stPacketHead. dwLength = dwPacketLen;     // 数据包的长度
pPacket->stPacketBody. dwPacketCount = 1;          // 包计数
// 复制协议内容到包体中
memcpy(pPacket->stPacketBody. bBodyContent,        // 复制协议内容
&stSellItem,
sizeof(SELL_ITEMS));
// 计算并重新设置 CRC 值
pPakcet->stPacketHead. dwCrc32 = CRC32(pPacket->stPacketBody);
```

经过上面的数据封装之后，接下来就是将 pPacket 指向的数据包发送出去。但是，发包通常不是简单地调用 send 函数，而是先对数据进行加密，然后才发送出去。

接下来给出一个高效的发包模型。通过对这个模型的分析，读者可以更好地理解断下 send 函数回溯定位 Call 方法的由来以及整个游戏的通信逻辑，如图 1-17 所示。可以看到，用户的每个动作在游戏客户端逻辑中都有一个类似 Switch 结构的 case 与之对应，可能的 Switch 如下。

```
switch(OP)
```

- case ——开启商店。

```
// 省略部分代码
break;
```

➢ case ——摆摊出售物品，步骤如下。

（1）收集数据。

（2）将数据复制到待发送缓存中（以包的形式）。

（3）读取待发送的包并进行加密。

（4）用加密后的包覆盖原始包。

（5）调用 send 函数发送加密后的包。

➢ case ——关闭商店。

```
// 省略部分代码
break;
```

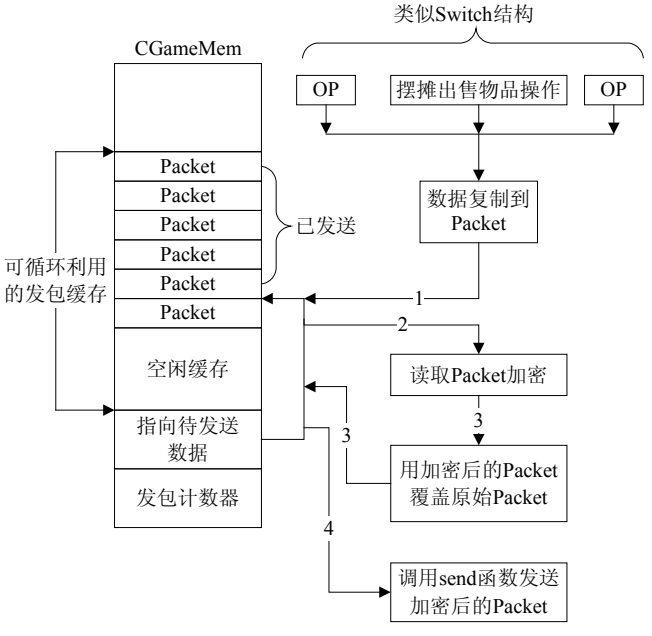


图 1-17　发包模型

每一个用户操作指令，最终都会变成一个被加密的包，存储在发包模型的可循环利用的缓存中，然后才发送出去。这个发包模型的优点如下。

➢ 统一了发包管理，例如，图 1–17 中的步骤 2、步骤 3、步骤 4 可以封装成一个发包出口函数。

➢ 可循环利用的缓存提高了内存的利用率。

但是，这个发包模型也有不足之处。从游戏安全的角度看，它的弱点在于比较容易从 send 函数进行回溯分析，从而定位 Switch 中的某个 case，进而挖掘出有价值的 Call。如果要对这个发包模型进行改正，可以将图 1–17 中步骤 2、步骤 3、步骤 4 的操作放在一个线程中专门处理，将步骤 1 的操作放在另外的线程中处理，线程间采用信号量或其他方式同步。这样，即使断下 send 函数来回溯，也不太容易定位。

当服务器端收到 ENUM_CMDPACKET_SELL_ITEMS 这个协议包的时候，就会为用户建立商店并摆上物品，所有这些都以数据的形式体现。下面以一幅简单的描述图来结束本节的协议之旅，如图 1–18 所示。

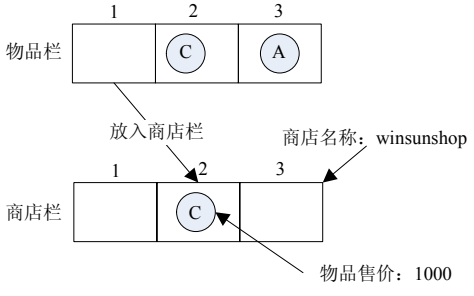


图 1-18 服务器端解封包后的数据操作

1.4.3 游戏内存对象布局

游戏中的各种对象，如玩家、怪物、武器、装备、精灵等，都以内存块的形式存在。其中，有些以结构体形式组织成块，负责保存对象的状态；有些以类的实例形式组织成块，在保存对象的同时提供对象的行为。所以，定位对象内存块的意义重大，不仅可以窥视对象的实例变量的内存布局，而且可以了解对象的行为。另外，修改玩家或怪物的 HP/MP 和速度、修改武器和装备的属性、召唤精灵等外挂行为，都需要定位对象的内存块。

本节将重点讨论如何定位对象内存块，以及常见对象内存块的内存布局和组织方式。

让我们从一个简化的 CPLAYER 类开始。

```
// CPLAYER 类的定义
 class CPLAYER
{
public:
    CPLAYER (DWORD dwHP, DWORD dwVector)
{ m_dwHP = dwHp; m_dwVector = dwVector;}  // CPLAYER 类的构造函数
    virtual DWORD GetVector(){return m_dwVector;}  // 获取速度
virtual void SetVector(DWORD dwVector){m_dwVector = dwVector;}
// 设置速度
virtual DWORD GetHP(){return m_dwHP;}              // 获取血量
virtual void SetHP(DWORD dwHP){m_dwHP = dwHP;}     // 设置血量
private:
    DWORD m_dwHP;           // 血量
    DWORD m_dwVector;       // 速度

};
```

上面的CPLAYER类比较简单：1个CPLAYER() 构造函数用于初始化血量及速度，4 个虚函数用于获取和设置血量及速度。接下来，我们新建一个 CPLAYER 对象，将它拖到 IDA 里，分析一下 CPLAYER 对象的内存布局，如图 1-19 所示。

```
.text:00401060 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401060 _main           proc near               ; CODE XREF: start+AF↓p
.text:00401060
.text:00401060 argc            = dword ptr  4
.text:00401060 argv            = dword ptr  8
.text:00401060 envp            = dword ptr  0Ch
.text:00401060
.text:00401060                 push    esi
.text:00401061                 push    0Ch             ; unsigned int
.text:00401063                 call    ??2@YAPAXI@Z    ; operator new(uint)
.text:00401068                 add     esp, 4
.text:0040106B                 test    eax, eax
.text:0040106D                 jz      short loc_40107E
.text:0040106F                 push    100
.text:00401071                 push    100
.text:00401073                 mov     ecx, eax
.text:00401075                 call    sub_401000      ; CPLAYER的构造函数，初始化HP和Vector
```

图 1-19　CPLAYER 对象

可以看到，“new”操作符后面紧跟着就是调用 CPLAYER 对象的构造函数 sub_401000。下面，让我们进入 sub_401000 函数，如图 1-20 所示。

```
.text:00401000 sub_401000      proc near               ; CODE XREF: _main+15↓p
.text:00401000
.text:00401000 arg_0           = dword ptr  4
.text:00401000 arg_4           = dword ptr  8
.text:00401000
.text:00401000                 mov     edx, [esp+arg_4]
.text:00401004                 mov     eax, ecx
.text:00401006                 mov     ecx, [esp+arg_0] ; ecx == this指针
.text:0040100A                 mov     dword ptr [eax], offset off_4050AC ; 虚表地址是4050AC
.text:00401010                 mov     [eax+4], ecx    ; 初始化m_dwHP
.text:00401013                 mov     [eax+8], edx    ; 初始化m_dwVector
.text:00401016                 retn    8
.text:00401016 sub_401000      endp
```

图 1-20　CPLAYER 对象的构造函数

图 1–20 中的 3 条指令如下。

- mov dword ptr[eax], offset off_4050AC
- mov [eax+4], ecx
- mov [eax+8], edx

通过这 3 条指令的赋值操作，可以分析出 CPLAYER 对象的内存布局，如图 1–21 所示。

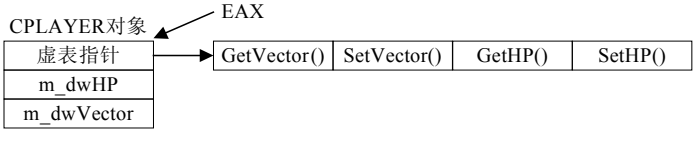


图 1-21　CPLAYER 对象的内存布局

根据图 1–21 所展示的 CPLAYER 对象的内存布局，我们可以很方便地修改 m_dwHP 和 m_dwVector，从而改变血量和速度。

从以上对 CPLAYER 对象的分析可以发现，如果一个类有构造函数，那么新建这个类的对象的时候，后面紧跟着的是这个类的对应构造函数的调用。同时，根据 IDA 的分析结果，构造函数能够暴露这个类的部分对象的内存布局。所以，要静态窥视类对象的内存布局，可以搜索“new”操作符，然后在它的后面挖掘构造函数，根据构造函数来分析对象内存布局。

在游戏的可执行文件（如 game.exe）中定位“new”操作符的方法大致有两种：一是通过 IDA 的静态反汇编，随机浏览反汇编代码，找到任意一个“new”操作符，通过这个“new”操作的交叉引用一次性全部定位“new”操作；二是通过动态 Hook 技术“HOOK new”调用。

在使用第二种方法之前，我们有必要了解一下如图 1–22 所示的 Windows 内存管理架构。

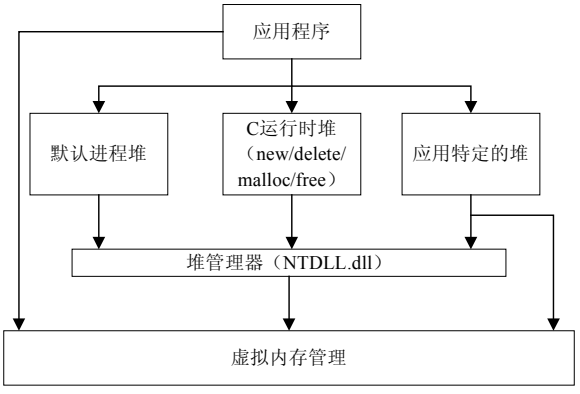


图 1-22　Windows 内存管理架构

可以看到，“new”和“malloc”都是 C 运行时堆，都调用底层的堆管理器函数来获取服务。如果读者想深入了解堆管理，可以参考 Mario Hewrdt 和 Daniel Pravat 的 *Advanced Windows Debugging* 一书中关于 Heap 的介绍。如果用 IDA 逆向分析 new 函数，我们将发现：new 函数会调用 msvcrt.dll 导出的 malloc() 函数，malloc() 函数继续调用 kernel32.dll 导出的 HeapAlloc() 函数，而 HeapAlloc() 函数其实就是 NTDLL.dll 导出的 RtlAllocateHeap() 函数。所以，如果要 Hook new 函数，可以使用 Hook malloc() 语句或 Hook HeapAlloc()/RtlAllocateHeap() 语句。不过，如果要 Hook HeapAlloc() 函数或 RtlAllocateHeap() 函数，我们就会发现，进程中对堆管理函数的频繁调用将干扰 Hook 的本意。关于这一点，在 Justin Seitz 的 *Gray Hat Python* 一书关于 Hooking 的部分也提到过，而且，该书还提供了用 hippie_easy.py 脚本来动态获取 RtlAllocateHeap() 函数的调用情况。不过，为了尽量不受干扰，我们可以直接 Hook malloc() 函数，而不深入 NTDLL 层，然后通过堆栈回溯来定位调用“new”操作符的返回地址。

上面所采用的分析方法，是通过“new”操作符后的构造函数来分析对象的部分内存布局的。当然，如果想知道此次截获的“new”操作针对的是角色对象、怪物对象、装备对象还是其他对象，就要结合差异分析的思想了。例如，我们可以在切换角色、切换地图、重新选怪或脱穿装备等行为的前后，看看是否有新的“new”调用。如果“new”调用发生，那么与引入新行为相关的“new”对象的操作肯定存在于这

个行为发生之后的“new”调用集合中，这可以帮助我们缩小分析范围。更详细的运用差异思想来分析外挂和游戏的实例，参见第 6.3 节。

差异思想在 Cheat Engine（作弊器）中也有丰富的体现，例如 Cheat Engine 的内存扫描（Memory Scan）功能。

在本节的最后，让我们看看如何运用差异思想来定位创建的新堆块。一个 NewHeapDlg 程序如图 1–23 所示。

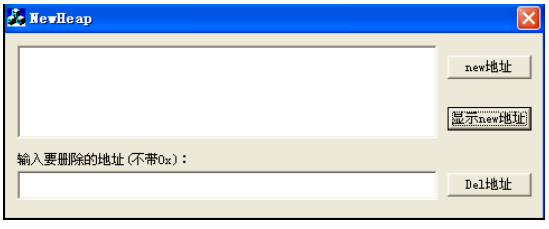


图 1-23　NewHeap 界面

NewHeapDlg 进程中堆块的分配信息如图 1–24 所示，我们可以看到通过 Heap32ListFirst、Heap32ListNext、Heap32First 和 Heap32Next 这 4 个 API 枚举出来的当前进程堆信息。

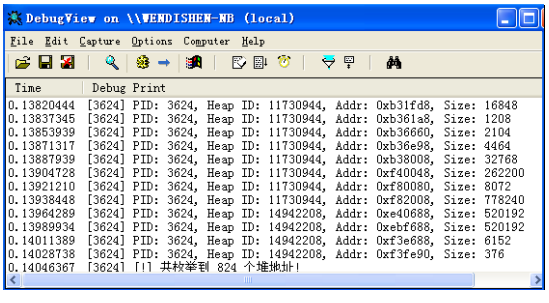


图 1-24　NewHeap 初始进程堆信息

如果单击“NewHeap”界面上的“new 地址”按钮，NewHeapDlg 程序将通过“new”操作符为堆分配一个新地址；如果单击“显示 new 地址”按钮，NewHeapDlg 程序将给出当前新分配的堆地址，如图 1–25 所示。这个时候，如果再次枚举 NewHeapDlg 进程堆，我们将看到如图 1–26 所示的堆信息。

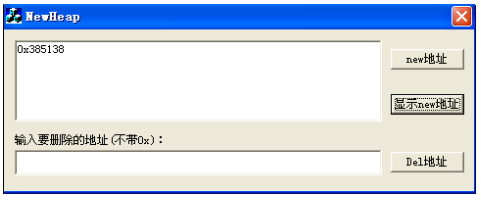


图 1-25　分配新地址

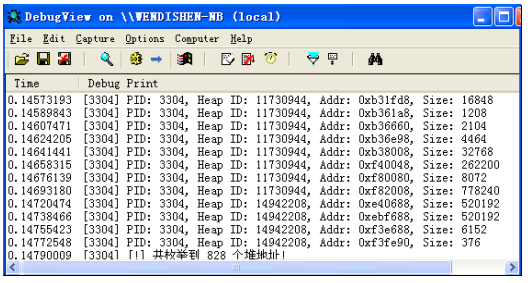


图 1-26　分配新地址后的进程堆信息

如图 1-24 所示，共枚举出 824 个堆块，而经过一次“new”操作后，如图 1-26 所示，共枚举出 828 个堆块。对比两次 dump 的进程堆信息，我们可以快速发现新建的堆内存。Cheat Engine 中有一项枚举进程堆块的信息，它位于“Memory View”→“View”→“Heaplist”菜单中。这个 Heaplist 只是动态跟踪进程当前已分配的堆块信息，并不像内存扫描那样多次枚举后对比差异。所以，我们既可以通过给 Cheat Engine 写插件来实现这种差异分析，也可以通过堆块枚举的 API 来实现这种差异分析。枚举堆块的代码大致如下。

```
        void EnumProcessHeaps()
    {
        BOOL   bHeapSuccess = FALSE;
        BOOL   bHeapListSuccess;
        HEAPENTRY32 stHeapEntry32 = {0};
       HEAPLIST32 stHeapList32 = {0};
        WORD      wHeapCounts = 0;
    HANDLE hHeapSnap = CreateToolhelp32Snapshot(TH32CS_SNAPHEAPLIST,
GetCurrentProcessId());
        if( INVALID_HANDLE_VALUE == hHeapSnap )
        {
            OutputDbgInfo(("[-]    EnumProcessHeap    CreateToolhelp32S
napshoterror!"));
            return;
        }
```

```
      stHeapList32.dwSize = sizeof(HEAPLIST32);
      bHeapListSuccess = Heap32ListFirst(hHeapSnap, &stHeapList32);
      if ( bHeapListSuccess == FALSE)
      {
         OutputDbgInfo(("[-] Heap32ListFirst error!"));
          return;
      }
      do
      {
          ZeroMemory(&stHeapEntry32, sizeof(stHeapEntry32));
          stHeapEntry32.dwSize  =  sizeof(HEAPENTRY32);
   bHeapSuccess                    =    Heap32First(&stHeapEntry32,
GetCurrentProcessId(), stHeapList32.th32HeapID);
          if ( bHeapSuccess == FALSE)
          {
             bHeapListSuccess      =      Heap32ListNext(hHeapSnap,
&stHeapList32);
              continue;
          }

          do
          {
              OutputDbgInfo(("PID: %d, Heap ID: %d, Addr: 0x%0x, Size:
%d", \
                             stHeapEntry32.th32ProcessID,
stHeapEntry32.th32HeapID, \
                               stHeapEntry32.dwAddress,
stHeapEntry32.dwBlockSize));
              wHeapCounts++;
              bHeapSuccess = Heap32Next(&stHeapEntry32);
          } while (bHeapSuccess == TRUE);
          bHeapListSuccess        =        Heap32ListNext(hHeapSnap,
&stHeapList32);
      } while (bHeapListSuccess == TRUE);
```

```
        OutputDbgInfo(("[!] 共枚举到 %d 个堆地址!", wHeapCounts));
    }
```

到目前为止，本书已经介绍了通过类构造函数来分析对象内存布局和通过动态 Hook“new”操作符或差异枚举堆块来定位关键内存块的方法。下面就让我们看看在游戏中一般如何组织和管理一些关键对象内存。

角色、怪物、物品等在游戏画面中出现的对象，一般会存放在一个较大的对象指针数组中，通过多级指针可以定位这个对象指针数组，对象之间通过对象类型字段和 ID 字段来区分。这些对象的大致组织方式如图 1–27 所示。

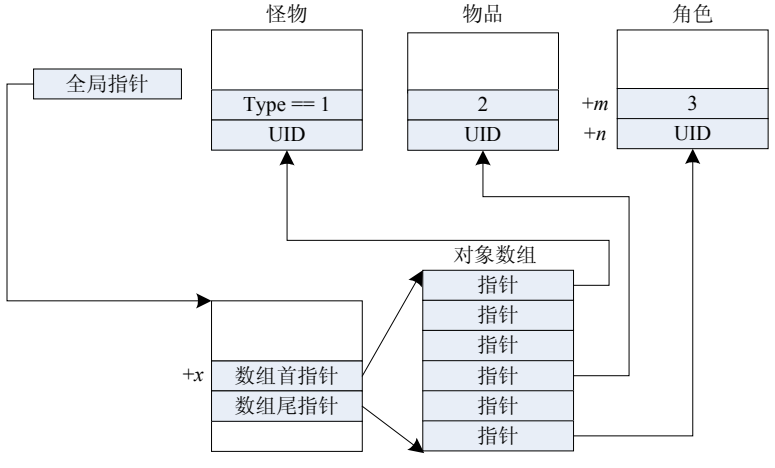


图 1-27　全局保存对象

可以看出，角色、怪物、物品等对象的地址都保存在一个对象数组中，它们偏移 *m* 和 *n* 的地方分别代表对象的类型和对象的 UID（全局唯一 ID）。通过类型我们可以知道这个对象是怪物对象还是物品对象，通过 ID 我们可以知道这个对象区别于其他对象的全局唯一标识。在游戏中，这个对象数组保存对象指针。根据以上信息，再结合 C++ 中的多态思想，我们就可以非常方便地在游戏的每一帧中通过遍历数组来更新对象的状态了。

对于外挂程序作者而言，获取这个对象数组非常重要。这个对象数组里保存了对象的地址，获取对象的地址再进行对象内存布局分析，就可以实现吸怪（与对象坐标有关）、加速等功能。那么，应该如何分析得出对象数组指针呢？我们可以对

任意一个对象地址下读断点，然后进行回溯分析，最终推导出这样一个公式——[[[*全局指针*]+*x*]+*y*]+*z*]。这个公式就代表了对象数组的首地址。

下面，再让我们看看游戏角色更换装备的时候，装备对象是如何在角色和物品栏之间切换的，如图 1–28 所示。

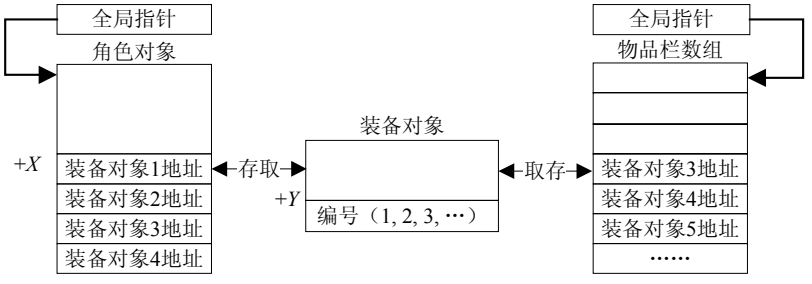


图 1-28　更换装备的过程

可以看到，角色对象偏移 *X* 的地方是一个拥有装备对象地址的数组，物品栏也是一个对象地址数组。当角色对象从物品栏选取某个装备的时候，根据装备对象偏移 *Y* 处提供的编号，相应装备对象的地址就会放入角色对象所对应的地址（角色对象基地址+*X*+编号 × 4）处，同时，物品栏数组的对应位置会被清空。当角色对象脱下某个装备的时候，角色对象所对应的存放装备对象地址的位置将被清空，同时，将该装备对象地址写入物品栏数组的任意位置。根据装备切换过程，我们可以运用 Cheat Engine 的内存扫描功能，当角色对象穿着装备的时候，扫描一道非零值，当角色对象脱掉装备的时候，扫描一道零值，一直扫描下去，很快就能定位存放装备对象的地址，同时定位装备对象的地址了。

1.5　外挂的设计思路

我们知道，安全是一个攻防对抗的过程，没有一方可以永远完全克制另一方，只能说在某一个时间段，其中的一方占据上风。因此，为了更好地防御和检测外挂，我们有必要深入了解一款优秀的外挂是如何制作的。只有掌握外挂的每一个制作环节和目的，才能更好地进行防御。在这一节中，就让我们一起从宏观上了解外挂的设计思路，其中谈到的每一个过程，后面都有对应的章节专门进行详细的阐述。

外挂启动之后，第一步当然是把自己的核心功能模块注入目标游戏进程，让游戏进程非自愿地加载外挂模块。这个步骤主要涉及 Windows 系统里的各种加载技术。本书的第 2 章会详细讨论每种加载技术及其优缺点，并会附上源代码。

第二步，当外挂模块进入游戏进程空间后，就需要隐藏外挂模块，免得被安全分析人员找出或被游戏安全模块检测到。本书的第 3 章会详细进行阐述——虽然都是 ring3 级的隐藏技术，但是非常实用和有效。

第三步，外挂隐藏之后，就可以开始和用户进行交互，以接收用户指令来进行特定的操作了。本书的第 4 章专门讲解如何构建一个安全的交互通道（列举了各种常见的交互通道），以及如何构建安全的交互以防止被安全分析人员定位。

第四步，外挂在接收用户的操作指令之后，就开始执行真正的与游戏相关的核心功能了，如吸怪、加速、释放技能、加红、加蓝等。当然，这些功能都需要具体的技术来支撑。本书的第 5 章和第 6 章将详细阐述支撑外挂功能的基本技术、Call 函数和 Hook，介绍的过程中会穿插一些外挂案例。

第五步，也是开发外挂和发挥外挂功能的前提，就是分析游戏的安全保护方案并绕过它。本书的第 8 章会详细讲解分析一款游戏的安全保护方案的思路。

第六步，也是最后一步，就是外挂的自我保护。本书的第 7 章主要讲解这部分内容，同时介绍一种简单却功效显著的方法。

当经过上面的 6 个步骤之后，一个完整的外挂程序才算制作完成。

第一人称射击类游戏的外挂制作过程与以上过程类似，只是在实现方案上可能略有差别，这将在本书的第 9 章中详细介绍。

在做到知己知彼之后，本书的最后一章，也就是第 10 章，会给出一套比较有效的外挂检测方案。

相信看过本书的人，无论是外挂程序作者，还是安全从业者，都会在脑海里对外挂这门技术有一个深刻的认识，从而更好地保护游戏的安全。

1.6 反外挂的思路

每款游戏的不同，带来的结果是每款游戏的反外挂方案和策略不同，因此，本节只介绍宏观和通用的反外挂思路。事实上，反外挂是一个系统工程，大致可以从

下面 4 个方向来进行安全防御。

1. 防止外挂程序作者分析游戏客户端代码

为了防止游戏客户端代码将有用的信息暴露给逆向分析人员，我们所能做的就是把各种调试信息、明文字符串等去掉，同时给客户端加上强有力的保护壳——最好是自己开发一套没有被别人分析过的保护壳。

2. 防止外挂模块注入游戏客户端

因为注入的途径非常多（本书第 2 章在注入方面有详细的讲解），所以防御方案可能也有很多，比较常用的就是采用双进程保护的启动方式来启动客户端，或者在驱动里添加监控加载模块的功能。

3. 防止外挂程序作者分析游戏通信协议

游戏客户端和服务器端的通信协议必须要加密，而且最好经常更换密钥或算法，以防止协议被模拟或篡改。当然，还要加上防止重放数据包的功能。

4. 防止外挂模块 Call 函数、修改代码或数据

这部分内容比较重要，本书的第 10 章会专门详细介绍。

1.7 本章小结

本章从游戏安全现状讲起，首先介绍什么是外挂，以及外挂对游戏的危害；接着，为了更好地讲解外挂和游戏的关系，分别讲述了游戏的 3 个核心概念，即游戏资源、协议和内存对象；最后，以阐述外挂和反外挂的思路来结束对游戏和外挂的介绍。

第 2 篇

外挂技术篇

第 2 章　五花八门的注入技术

了解了外挂与游戏进程间的寄生关系之后，接下来我们就要讨论外挂如何寄生，或者说外挂注入游戏进程空间的方法了。

在安全领域，“注入”是一个曝光率很高的词，就像子弹射进物体一样，最终目的是让注入者留在注入对象“体内”。从技术角度来看，注入其实就是迫使第三方进程非自愿加载某个模块，所以本章采用“加载”一词来代表“注入”。在反外挂的过程中，注入和反注入一直处在持续不断的、白热化的对抗中。

本章将罗列一些加载方式，同时结合笔者在对抗中的经验，给出一些优缺点的点评。

2.1　注册表注入

在 Windows NT/2000/XP/2003 操作系统中，当需要加载 user32.dll 的程序启动时，user32.dll 会加载注册表键 HLM\Software\Microsoft\WindowsNT\CurrentVersion\Windows\AppInit_Dlls 下面列出的所有模块。

根据这个原理，外挂可以将外挂模块所在的路径写到 AppInit_Dlls 键下，待游戏进程启动并将外挂模块带入之后，再删除 AppInit_Dlls 键的值以清除痕迹，其核心函数如下，具体代码参见本章资源包中的 RegInject 工程代码。

```
    // 定义键值
    #define DSTKEY "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\
Windows"
    // 打开主键
    RegOpenKeyEx(
                HKEY_LOCAL_MACHINE,
                DSTKEY,
                0,
                KEY_ALL_ACCESS,
                &hKey);
    // 设置 AppInit_DLLs 键的值，其中“cDllPath”为待注入 DLL 的路径
    RegSetValueEx(
           hKey,
           "AppInit_DLLs",
           0,
           REG_SZ,
           cDllPath,
           strlen((char*)cDllPath)+1
          );
```

注册表注入的优点和缺点分析如下。

- 优点：简单，易于实现。
- 缺点：第一，系统重启后才能实现注入，且对 DLL 的稳定性要求较高。建议只在虚拟机里试用，因为 DLL 写得不好会造成 BSOD（蓝屏），连安全模式也进不去。第二，易于被像 ProcessMonitor 这样的用于监测注册表操作的软件记录，进而被安全分析人员发现。

2.2　远线程注入

远线程注入的核心思想是利用 Windows 提供的远线程机制，在目标进程中开启一个加载模块的远线程，使外挂模块被该远线程加载到游戏的地址空间。

远线程使用的关键 API 有 WriteProcessMemory、CreateRemoteThread 和 LoadLibrary，它们的声明如下。

```
BOOL WriteProcessMemory(
    HANDLE hProcess,
    LPVOID lpBaseAddress,
    LPVOID lpBuffer,
    DWORD nSize,
    LPDWORD lpNumberOfBytesWritten
  );
```

WriteProcesssMemory 参数的说明如下。

- hProcess：远进程句柄。
- lpBaseAddress：远进程待写地址。
- lpBuffer：本进程空间 buffer 地址。
- nSize：lpBuffer 所指空间的大小。
- lpNumberOfBytesWritten：返回实际写入远进程的字节数。

```
HANDLE CreateRemoteThread(
       HANDLE hProcess,
       LPSECURITY_ATTRIBUTES lpThreadAttributes,
       SIZE_T dwStackSize,
       LPTHREAD_START_ROUTINE lpStartAddress,
       LPVOID lpParameter,
       DWORD dwCreationFlags,
       LPDWORD lpThreadId
       );
```

CreateRemoteThread 参数的说明如下。

- hProcess：远进程句柄。
- lpThreadAttributes：线程安全描述字，指向 SECURITY_ATTRIBUTES 结构的指针。
- dwStackSize：线程栈大小，以字节为单位表示。
- lpStartAddress：一个 LPTHREAD_START_ROUTINE 类型的指针，指向在远程进程中执行的函数地址。
- lpParameter：传入参数。

- dwCreationFlags：创建线程的其他标志。
- lpThreadId：线程 ID。如果为 NULL，则不返回。

```
HMODULE WINAPI LoadLibrary(
  __in  LPCTSTR lpFileName
);
```

LoadLibrary 参数的说明如下。

- lpFileName：待加载模块的文件路径。

远线程加载的成功基于下面两个因素。

- Win32 API 中的线程函数和 LoadLibrary 函数的原型声明本质上一致。在 Win32 API 中，线程函数的原型是 DWORD ThreadFunc(PVOID pThreadParam)，对比 LoadLibrary 的函数原型，返回类型都占据 4 字节，参数都只有 1 个，而且是占据 4 字节的指针，所以，两个函数的原型声明是等同的，可以把 LoadLibrary 函数的地址当作线程函数的地址传给 CreateRemoteThread 函数的参数 lpStartAddress，同时把要加载的模块路径传给 lpParameter 参数。
- Win32 系统中存放 LoadLibrary 函数地址的 kernel32.dll 在各进程中的加载地址一致。基于这个事实，就可以在本进程中通过 GetProcAddress 函数获取 LoadLibrary 的地址，而不用想办法到远进程空间获取 LoadLibrary 的地址了。

基于上面两个因素，远线程加载流程的简化图如图 2-1 所示，步骤如下。

（1）调用 WriteProcessMemory 函数将外挂模块的文件路径写入游戏进程空间。

（2）调用 CreateRemoteThread 函数开启游戏进程的 LoadLibrary 远线程，以加载外挂模块。

因为远线程加载方式比较古老，稍微具备反外挂机制的系统都已经防御了此法，所以在这里不做过多阐述，具体代码参见本章资源包中的 RemoteThreadInject 工程。

远线程加载的优点和缺点如下。

- 优点：简单，易实现。
- 缺点：容易被监控。只要监控本进程对 LoadLibrary 函数的调用，就可以很好地防御远线程加载。

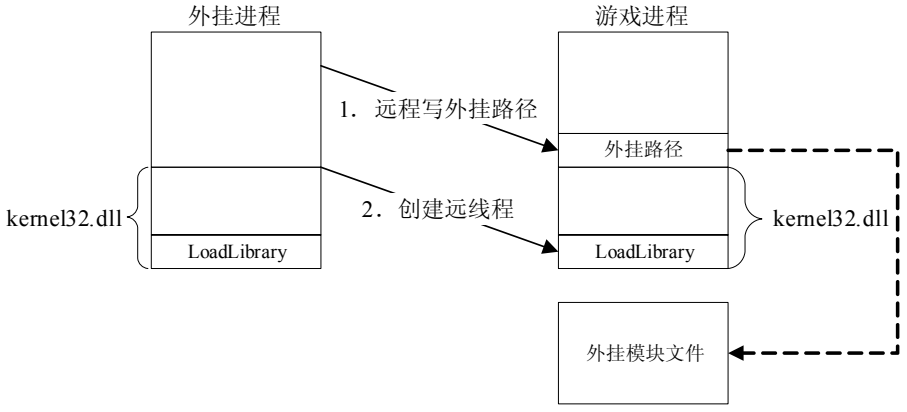


图 2-1　远线程加载流程简化图

2.3　依赖可信进程注入

为了躲避反外挂系统的扫描和分析人员的分析，有些时候，依赖可信任第三方进程所进行的转注入，可能会取得比较好的效果。曾经有一款加载方式比较新颖的外挂，就是采用这种方式来实现注入的。现在，就让我们通过分析这款外挂的加载方式来谈谈转注入。

我们都知道，services.exe 进程是 Windows 操作系统的重要组成部分，它管理着服务的启动和停止，同时，它也是许多木马关注的宿主。这里再次强调：木马不是用户主动使用的，而是被动接受的，但外挂是用户主动使用的，所以，木马不仅要突破安全机制的防御，还要突破游戏程序的防御。从突破技术上看，木马考虑的问题比外挂多，使用外挂的用户往往会为了运行外挂程序而暂停使用安全产品，这是造成游戏木马泛滥的一个原因。

现在，让我们回到 services.exe 进程上来。不知道有没有读者用 OllyDbg 调试过 services.exe 进程。如果调试过的话，那要“恭喜”你了——幸运的则会附加不上，不幸的则会看到一个对话框，告诉你距离关机还有多长时间。所以，通过木马和外挂找 services.exe 进程是一个非常明智的选择——因为 services.exe 不易调试、权限高、隐蔽性好。

假设外挂进程为 Bot.exe，外挂主模块为 FQ.dll，游戏进程为 Game.exe，这种转

注入的步骤如下。

（1）Bot.exe 调用 WriteProcessMemory 和 CreateRemoteThread 函数，采用远线程注入方式将 FQ.dll 注入 services.exe 进程。Bot.exe 调用 Sleep(35000) 函数，使自己进入睡眠状态。

（2）services.exe 中的 FQ.dll 调用 CreateToolhelp32Sanpshot 和 Process32Next 来遍历进程，判断 Game.exe 进程是否启动。如果发现 Game.exe 启动，就采用远线程或其他方式将 FQ.dll 注入 Game.exe 进程；如果未发现 Game.exe 启动，则继续循环遍历。

（3）Bot.exe 进程睡眠状态结束，开始下一步。补充一句：按照常理，35 000ms 已经足够 services.exe 进程中的 FQ.dll 实现对 Game.exe 进程的注入了。

（4）Bot.exe 调用 CreateToolhelp32Sanpshot 和 Process32Next 来遍历进程，寻找 services.exe 进程。找到该进程后，遍历其中的模块，找到 FQ.dll。

（5）Bot.exe 进程通过调用 CreateRemoteThread 远线程来调用 FreeLibrary，将 services.exe 中的 FQ.dll 从 services.exe 进程中卸载。

经过上面这巧妙的 5 步操作，FQ.dll 悄无声息地从 Bot.exe 进入 services.exe，然后进入 Game.exe。services.exe 进程在整个注入过程中只起中转作用。

图 2–2 用简化的形式描述了转加载的过程。

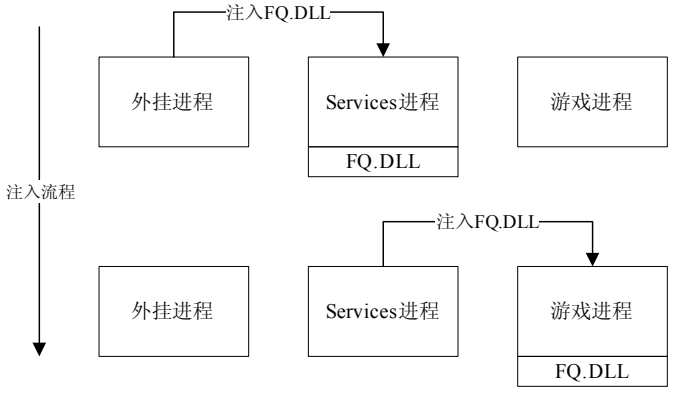


图 2-2　转加载流程简化图

依赖可信进程注入的优点和缺点如下。

- 优点：隐蔽性好，不易被分析和发现。

- 缺点：遍历进程的方法需要改进，以防止被假进程所迷惑；显式调用 API 也不妥。

2.4 APC 注入

APC（Asynchronous Procedure Call，异步过程调用）是在一个特定线程环境下被异步执行的函数，分为用户模式 APC 和内核模式 APC。每个线程都有一个 APC 队列。在用户模式下，当线程调用 SleepEx、WaitForSingleObjectEx 等进入“Alterable Wait Status”状态（可警告的等待状态）的时候，系统就会遍历该线程的 APC 队列，然后按照先进先出的顺序来执行这些 APC。APC 的具体介绍参见 http://msdn.microsoft.com/en-us/library/ms681951(v=VS.85).aspx。

在用户模式下，微软提供了 QueueUserAPC 这个 API 来向一个线程插入 APC。下面是 QueueUserAPC 的声明。

```
DWORD WINAPI QueueUserAPC(
  __in  PAPCFUNC pfnAPC,
  __in  HANDLE hThread,
  __in  ULONG_PTR dwData
);
```

- pfnAPC：指向一个 APC 函数。
- hThread：将要插入 APC 的线程句柄。
- dwData：APC 函数的参数。

下面给出一段 APC 加载的伪代码来分析一下加载步骤。

```
// 1. 以 suspend 方式创建待加载的目标进程
if(CreateProcess(sProcName,NULL,NULL,NULL,FALSE,CREATE_SUSPENDED
,NULL,NULL,&st,&pi))
{
// 2. 目标进程地址空间分配待加载的 DLL 路径空间
lpDllName = VirtualAllocEx(pi.hProcess, NULL, (strlen(sDllName) +
1), MEM_COMMIT, PAGE_READWRITE);
```

```
    // 3. 把待加载的 DLL 路径写入目标进程空间
    if(WriteProcessMemory(pi.hProcess,   lpDllName,   sDllName,strlen
(sDllName), NULL))
    {
      // 4. 获取 LoadLibrary 的地址
    LPVOID                                                          =
nLoadLibrary=(LPVOID)GetProcAddress(GetModuleHandle("kernel32.dll"),
    "LoadLibraryA");
    // 5. 调用 QueueUserAPC 向远线程插入一个 APC，这个 APC 就是 LoadLibrary
        if(!QueueUserAPC((PAPCFUNC)nLoadLibrary,pi.hThread,(ULONG_
PTR)lpDllName))
       {
           OutputDebugString("[-] APCInject QueueUserAPC call  error!");
           dRet=-6;
        }
    }
    }
```

当然，除了可以通过 VirtuaAllocEx 和 WriteProcessMemory 函数在目标进程地址空间存放待加载 DLL 的路径，我们还可以通过 CreateFileMapping、MapViewOfFile 和 NtMapViewOfSection 函数的组合来映射路径，具体参见本章资源包中的 APCInject 工程。这些代码是在 Visual Studio 2005 下编译通过的，且只能在某些环境下起作用。通过 ApcInject 程序将 WaiGuaDll.dll 加载到 IPMSG.exe 程序中的过程如图 2-3 所示。

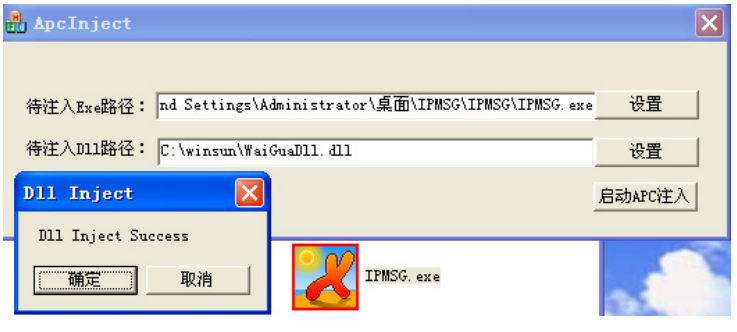


图 2-3　ApcInject

APC 注入的优点和缺点如下。

- 优点：比较隐蔽，简单。
- 缺点：实现的条件比较苛刻。

2.5 消息钩子注入

SetWindowsHookEx 函数是微软提供给程序开发人员进行消息拦截的一个 API。不过，它的功能不仅可以用作消息拦截，还可以进行 DLL 注入。

SetWindwosHookEx 的原型声明如下。

```
HHOOK SetWindowsHookEx(
 Int idHook,
 HOOKPROC lpfn,
 HINSTANCE hMod,
 DWORD   dwThreadId
);
```

- idHook：指示将要安装的挂钩处理过程的类型。例如，idHook 为“WH_CALLWNDPROC”时代表安装一个挂钩处理过程，在系统将消息发送至目标窗口处理过程之前对该消息进行监视。
- lpfn：指向相应的挂钩处理过程。
- hMod：指示了一个 DLL 句柄。该 DLL 包含参数 lpfn 所指向的挂钩处理过程。
- dwThreadId：指示了一个线程标识符，挂钩处理过程与线程相关。若此参数值为 0，则该挂钩处理过程与所有现存的线程相关。

如果要去掉消息钩子，可以调用 UnhookWindowsHookEx 函数。

由于 SetWindowsHookEx 注入与 Windows 的消息处理流程相关，下面就让我们先简单了解一下 Windows 的消息处理流程，再去看看 SetWindowsHookEx 的注入流程。

如图 2-4 所示，当用户进行窗口操作的时候，产生的消息先被 Windows 集中到系统消息队列中，然后转发到线程消息队列中。线程消息分发逻辑通过 GetMessage 函数从消息队列取中出消息，经过 TranslateMessage 函数的转化，调用 DispatchMessage 函数，以异步方式将消息送到窗口回调函数 WinProc 中进行处理。

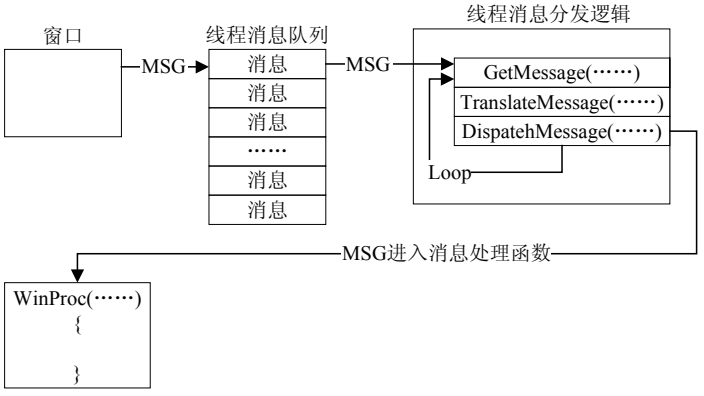


图 2-4　Windows 消息处理的简化流程

Windows 提供 SetWindowsHookEx 这个 API 的本意是希望在 MSG 被窗口处理过程 WinProc 处理之前，给开发者一个感知消息的机会。这种分层设计比较灵活，类似系统底层分层驱动之间的 IRP 传递机制。

接下来，让我们看看利用 SetWindowsHookEx 函数在 WinProc 处理消息之前插入一个消息处理的过程，以及截获、处理和传递 MSG 的过程，如图 2–5 所示。

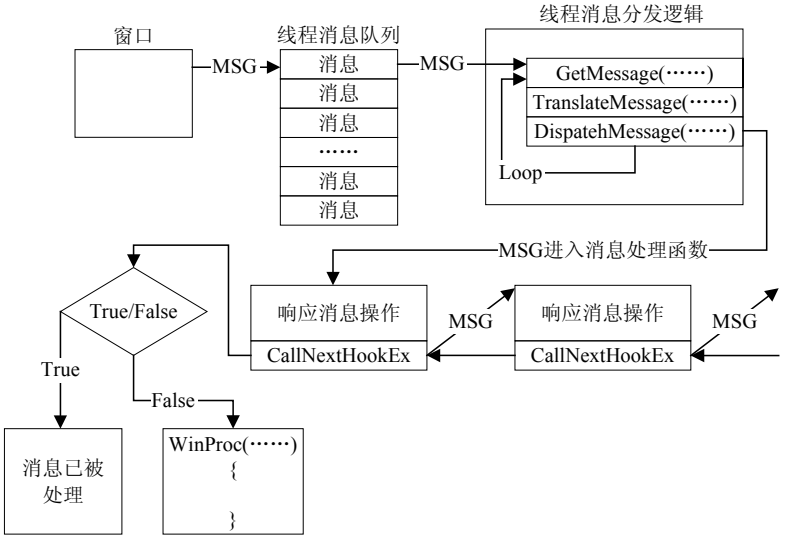


图 2-5　SetWindowsHookEx 插入消息拦截过程后的消息传递流程

可以看出，MSG 被新的消息处理过程接收，然后通过微软提供的另一个 API，即 CallNextHookEx 函数，向下传递 MSG。第一个 CallNextHookEx 函数如果返回“True”，表示这个 MSG 已经被处理；如果返回“False”，系统将继续把 MSG 传递给 WinProc 进行处理。

图 2-4 只反映了 MSG 在调用 SetWindowsHookEx 函数前后消息处理流程的变化，事实上，我们忽略了操作系统在这个过程中的一个关键行为——当调用 GetMessage 函数获取消息的时候，系统会判断是否要对该消息安装消息钩子，以及安装了消息钩子的模块是否要加载到本进程空间，如果没有加载则进行加载。这里的加载就是我们需要的注入。

下面是注入的核心代码。

```
// 利用 Windows API SetWindowsHookEx 实现注入 DLL
BOOL SetWinHKInject(char * pszDllPath, char * pszProcess)
{
// 加载待注入的 DLL 到本进程空间
    hMod = LoadLibrary(pszDllPath);
    if(!hMod)
    {
        OutputDebugString("[+] LoadLibrary error!\n");
        goto Exit;
    }
   // 获取待注入 DLL 中导出的消息钩子过程函数的地址
   lpFunc = (DWORD)GetProcAddress(hMod, "MyMessageProc");
    if(!lpFunc)
    {
        OutputDebugString("[+] GetProcAddress error!\n");
        goto Exit;
    }
    // 获取待注入进程的线程 ID
    dwThreadId = GetTargetThreadIdFromProcname(pszProcess);
    if(!dwThreadId)
     goto Exit;
    // 调用 SetWindowsHookEx 实现消息钩子注入
```

```
        g_hhook = SetWindowsHookEx(
                    WH_GETMESSAGE,//WH_KEYBOARD,//WH_CALLWNDPROC,
                    (HOOKPROC)lpFunc,
                    hMod,
                    dwThreadId
                  );
    Exit:
       if(hMod)
          FreeLibrary(hMod);
        return bSuccess;

    }
    // 待注入 DLL 导出的消息钩子函数 MyMessageProc
    __declspec(dllexport) LRESULT MyMessageProc(int code, WPARAM wParam,
LPARAM lParam)
    {
      //
      // 开发人员对消息的处理
      //
      return CallNextHookEx(g_hhook, code, wParam, lParam);
    }
```

具体代码参见本章资源包中的 MsgInject 工程。这里涉及 MsgInject 和 MsgWaiGua 这两个工程的代码，它们分别是实现注入的 EXE 和负责消息传递的 DLL。

消息钩子注入的优点和缺点如下。

- 优点：代码和原理简单，容易实现。
- 缺点：容易被发现和防御。

2.6　导入表注入

导入表是 Windows PE 文件中的一组数据结构，可执行程序（即 EXE）被加载到地址空间后，每个导入的 DLL 模块都有一个对应的导入表，PE 加载器会根据导入表来加载进程需要的其他 DLL 模块。

导入表的数据结构如下。

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
 union {
      DWORD   Characteristics;
      DWORD   OriginalFirstThunk;
    };
 DWORD   TimeDateStamp;
 DWORD   ForwarderChain;
 DWORD   Name;
 DWORD   FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR;
```

- OriginalFirstThunk/Characteristics：指向导入名称表（INT）的 RVA（相对虚拟地址）。INT 是一个 IMAGE_THUNK_DATA 结构的数组，数组中的每个元素指向一个 IMAGE_IMPORT_BY_NAME 结构，INT 以元素 0 结束。
- TimeDateStamp：时间戳，可以忽略。
- ForwarderChain：如果没有前向引用（forwarders）的话就是 –1。
- Name：被导入 DLL 的名字指针，是一个 RVA。
- FirstThunk：指向导入表（IAT）的 RVA。IAT 是一个 IMAGE_THUNK_DATA 结构的数组。

导入表的逻辑结构如图 2–6 所示。

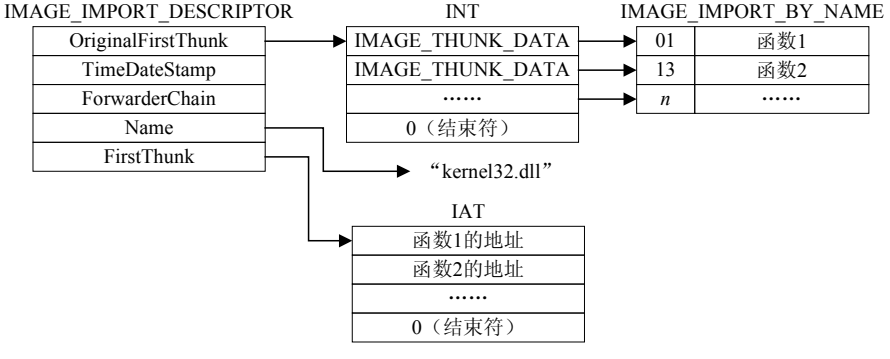
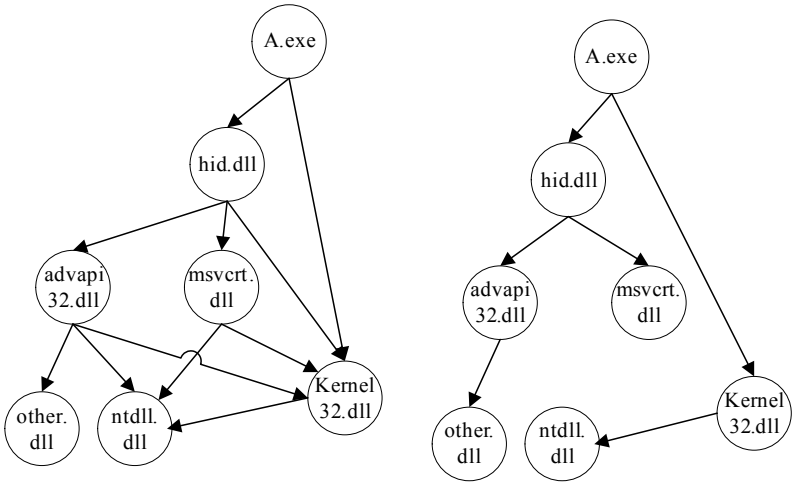


图 2-6 导入表的逻辑结构

关于导入表的细节，读者可以参考《加密与解密（第 3 版）》或《黑客反汇编揭秘》等书。

Windows 中的 PE 加载器，就是根据图 2–6 中的 IMAGE_IMPORT_DESCRIPTOR 数组描述的欲导入模块信息来加载对应模块的。

现在，让我们从另外一个角度来观察模块的导入关系。假设有一个可执行程序 A.exe，运行之后，其进程空间模块有 hid.dll、msvcrt.dll、advapi32.dll，基础模块有 ntdll.dll、kernel32.dll，代表其他模块的是 other.dll，它们的导入关系如图 2–7 所示。



A. 模块导入依赖关系　　B. 简化的模块导入依赖关系

图 2-7　模块的导入关系

图 2–7 中的圆形代表进程空间的模块，带箭头的指引线代表导入的依赖关系。例如，在图 2–7（A）中，进程 A.exe 导入了 hid.dll 和 kernel32.dll，而 hid.dll 又导入了 advapi32.dll、msvcrt.dll 和 kernel32.dll，kernel32.dll 不仅被 A.exe 模块导入，也被 hid.dll 模块导入。但是，因为一个进程空间只需要加载一个 kernel32.dll 模块，所以，图 2–7（B）对图 2–6（A）进行了简化，只要有两个箭头指向的模块，都可以只保留一个箭头指向。简化后的模块导入依赖关系图看上去比较清晰和简洁，很像一棵树，树枝代表了图 2–6 中的 IMAGE_IMPORT_DESCRIPTOR 结构。Windows 的 PE 加载器就是根据这种依赖关系来加载模块的。

在这里读者可能会问：这跟注入有什么关系呢？其实，注入就是为了让自己的模块进入其他进程空间，既然 PE 加载器能根据图 2–7 中的箭头指向来加载模块，难

道我们不能把一个节点“挂”到这棵“树”上吗？答案显然是肯定的，而这就是本节所讨论的导入表注入。

一般情况下，导入表的加载都是通过修改进程、导入依赖树上某个模块的导入表来实现的（即导入依赖树上无故增加一个分支节点），而这个修改动作是在进程加载该模块之前完成的（即修改模块文件中的导入表）。待外挂模块被加载到进程空间之后，我们可以将被修改的模块文件改回来，以防止文件修改操作被发现。

笔者分析过的一个系列外挂就是采用这种加载方式注入的，而且这个系列外挂为了躲避检测，轮番在导入依赖树上寻找挂靠节点。这个系列外挂中的某款外挂会选择系统目录下的 hid.dll 模块作为挂靠节点。

下面我们分别从节表和导入表两个方面，对原始 hid.dll 文件和被外挂修改的 hid.dll 文件进行对比，以便了解外挂是如何制作出来的。

系统目录下原始 hid.dll 的节表信息如图 2-8 所示。根据 PE Editor 显示的结果，hid.dll 只有 4 个节，分别是.text、.data、.rsrc、.reloc。

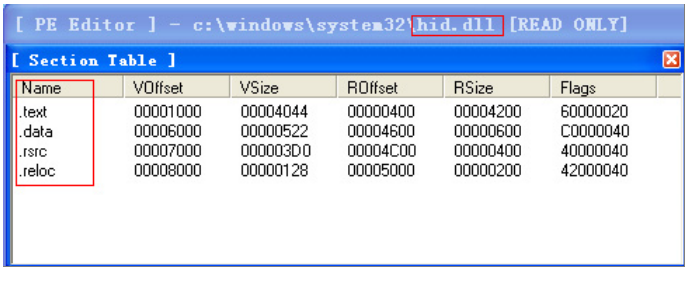


图 2-8 原始 hid.dll 文件的节表信息

系统目录下原始 hid.dll 的导入表如图 2-9 所示。根据 PE Editor 显示的结果，hid.dll 只导入了 advapi32.dll、kernel32.dll 和 msvcrt.dll 这 3 个模块。

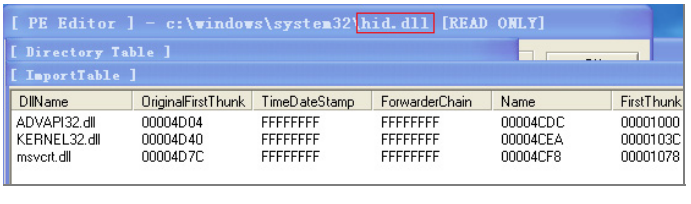
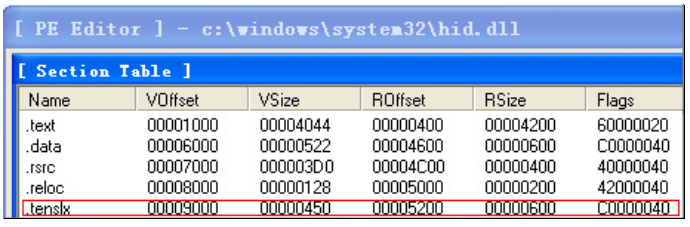


图 2-9 原始 hid.dll 文件的导入表

下面再让我们看看经过外挂修改的 hid.dll 文件中的节表和导入表是什么样的。

对比图 2-8 和图 2-10，可以看到，外挂在节表的末尾增加了一个叫做 .tenslx 的节，这个节的起始 RVA 是 0x9000。这代表了什么呢？看看外挂对导入表的修改就会明白了。

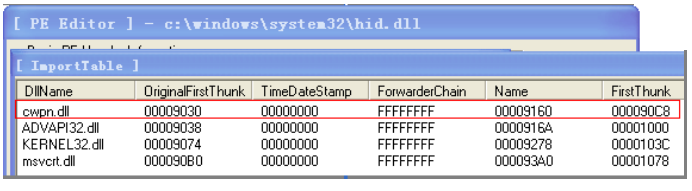

[PE Editor] - c:\windows\system32\hid.dll

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	00004044	00000400	00004200	60000020
.data	00006000	00000522	00004600	00000600	C0000040
.rsrc	00007000	000003D0	00004C00	00000400	40000040
.reloc	00008000	00000128	00005000	00000200	42000040
.tenslx	00009000	00000450	00005200	00000600	C0000040

图 2-10　被修改的 hid.dll 的节表

对比图 2-9 和图 2-11，可以看到，外挂在导入表的数组中增加了一个导入项，那就是 cwpn.dll，这是外挂的模块，而且 cwpn.dll 和其他 3 个模块的 INT（如图 2-9 所示）的 RVA 都大于 0x9000。现在读者应该明白外挂为什么会专门增加一个起始 RVA 在 0x9000 的节 .tenslx 中了吧！


[PE Editor] - c:\windows\system32\hid.dll

[ImportTable]

DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
cwpn.dll	00009030	00000000	FFFFFFFF	00009160	000090C8
ADVAPI32.dll	00009038	00000000	FFFFFFFF	0000916A	00001000
KERNEL32.dll	00009074	00000000	FFFFFFFF	00009278	0000103C
msvcrt.dll	000090B0	00000000	FFFFFFFF	000093A0	00001078

图 2-11　被修改的 hid.dll 的导入表

外挂的工作思路如下。

- 在原始 hid.dll 的最后一个节后面增加一个 .tenslx 节，其作用是存放原始的导入表和新增的导入项。
- 将原始的导入表数组和新增的导入项重新构建，放置到新节 .tenslx 中。

上面这些操作用 PE Editor 来实现还是很方便的，不过，写代码非常考验我们对 PE 格式的熟悉程度，最近市面上有一本关于 PE 的书——《Windows PE 权威指南》，相信会对读者了解 PE 有所帮助。

因为导入表加载的原理就是在原有的 IMAGE_IMPORT_DESCRIPTOR（以下简称

IID）数组中增加一条 IID 记录，所以我们必须考虑原来存放 IID 的位置是否有空间容纳这个新的 IID。为了避免这种意外的发生，有两种方法可以采用：一是在所有的节中寻找一块能容纳新 IID 数组的文件空隙；二是新增一个足够大的节来专门存放新的 IID 数组。在本节中，我们会采用第二种方法——新增节，并把新的 IID 数组放入该节。

一个简化了的 PE 文件格式，如图 2-12 所示。

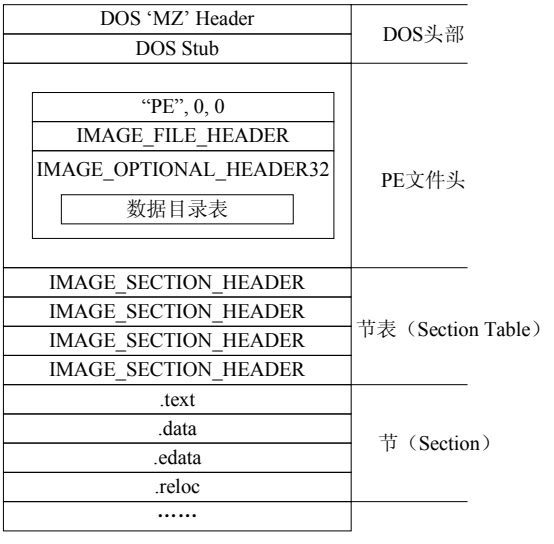


图 2-12　简化的 PE 文件格式

由图 2-12 我们知道，可以在文件的末尾增加一块空间作为新节。例如，节的名字叫做“WINSUN”，那么根据 PE 格式的定义，就必须对应增加一个 IMAGE_SECTION_HEADER 用来描述新增节“WINSUN”所在的文件偏移和 RVA 等信息。

在本章的资源包中，ImportTableInject 工程的 AddNewSection 函数实现了增加新节的功能，该函数的核心代码如下。

```
BOOL AddNewSection(LPCTSTR lpStrModulePath, DWORD dwNewSectionSize)
{
    // 1、调用 CreateFile，CreateFileMapping，MapViewOfFile
    // 将 lpStrModulePath 指向的 PE 文件映射到本进程
```

```
    lpMemModule = MapViewOfFile(hFileMapping, FILE_MAP_ALL_ACCESS, 0, 0,
    dwFileSize);
    // 2、判断是否是 PE 文件
       if (((PIMAGE_DOS_HEADER)lpData)->e_magic != IMAGE_DOS_SIGNATURE)
       {
                OutputDebugString("[-]  AddNewSection  PE  Header  MZ
error!\n");
                goto _EXIT_;
       }
    pNtHeader  =  (PIMAGE_NT_HEADERS)(lpData  +  ((PIMAGE_DOS_HEADER)
(lpData))->e_lfanew);
    if ( pNtHeader->Signature != IMAGE_NT_SIGNATURE )
    {
                OutputDebugString("[-]  AddNewSection  PE  Header  PE
error!\n");
                goto _EXIT_;
       }
       // 3、判断是否可以增加一个新节表 IMAGE_SECTION_HEADER
       if ( ((pNtHeader->FileHeader.NumberOfSections + 1) *
    sizeof(IMAGE_SECTION_HEADER)) > (pNtHeader->OptionalHeader.SizeOf
Headers))
       {
                OutputDebugString("[-] AddNewSection cannot add a new
section!\n");
                goto _EXIT_;
       }
      // 4、获取新增节的文件偏移
    pNewSection  = (PIMAGE_SECTION_HEADER)(pNtHeader+1) +
    pNtHeader->FileHeader.NumberOfSections;

    // 5、填充新节表
    memcpy(pNewSection->Name, "WINSUN", strlen("WINSUN"));
    pNewSection->VirtualAddress = voffset;
    pNewSection->PointerToRawData = roffset;
    pNewSection->Misc.VirtualSize = vsize;
```

```
    pNewSection->SizeOfRawData = rsize;
    pNewSection->Characteristics = IMAGE_SCN_MEM_READ |
    IMAGE_SCN_MEM_WRITE;
            // 6、修改 IMAGE_NT_HEADERS，增加新节表
            pNtHeader->FileHeader.NumberOfSections++;
            pNtHeader->OptionalHeader.SizeOfImage += vsize;
            // 7、增加新节到文件尾部
            SetFilePointer(hFile,0,0,FILE_END);
            PBYTE pbNewSectionContent = new BYTE[rsize];
            ZeroMemory(pbNewSectionContent, rsize);
            bSuccess = WriteFile(hFile, pbNewSectionContent, rsize,
&dwWriteBytes, NULL);
    }
```

AddNewSection 函数的详细代码参见本章资源包中 ImportTableInject 工程下的 ImportTableInjectDlg.cpp 文件。

有了新节以后，我们就可以把原来的 IID 数组移入，同时在其后新增一个 IID。

新增一个 IID 需要更新与 IID 相关联的各项信息。与 IID 相关联的各个数据结构之间的关系如图 2-13 所示。

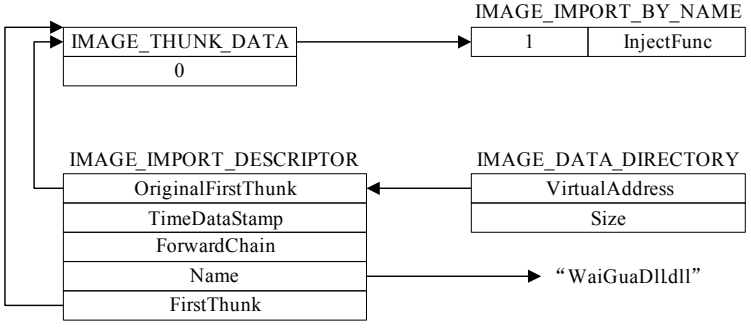


图 2-13 IID 关联数据结构关系

根据图 2-13，首先构建 IMAGE_IMPORT_BY_NAME（简称 IIBN），然后依次构建 IMAGE_THUNK_DATA（简称 ITD）、IMAGE_IMPORT_DESCRIPTOR（简称 IID）和 IMAGE_DATA_DIRECTORY（简称 IDD），最后把他们串联起来，就可以新增一个

IID 了。在新增的“WINSUN”节中，各个数据结构的布局如图 2-14 所示。

新节“WINSUN”

原有的IID	ITD	0	新增的IID	WaiGuaDll.dll	IIBN

图 2-14　IID 关联数据结构文件布局

在本章资源包 ImportTableInject 工程下的 ImportTableInjectDlg.cpp 文件中，Add NewImportDescripto 函数实现了在新节中增加导入表的功能，具体代码读者可以结合图 2-14 来看，这样比较容易理解。

关于导入表加载的例子，本书提供了一个演示程序 ImportTableInject.exe 放在本章资源包的 ImportTableInject 工程下，如图 2-15 所示。

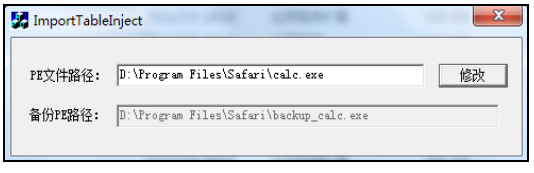


图 2-15　导入表加载示例程序

图 2-15 中测试的是 Windows 的自带程序 calc.exe。经过修改之后，通过 LordPE 可以看到新增的导入 WaiGuaDll.dll 的项目，如图 2-16 所示。

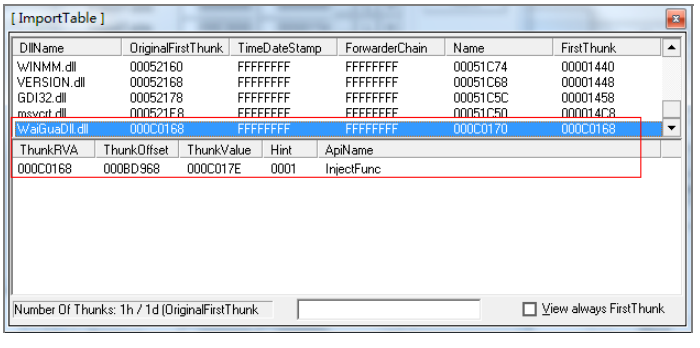


图 2-16　calc.exe 的导入视图

这个时候，只要 calc.exe 目录下含有 WaiGuaDll.dll 模块，就可以完成加载了。本章的资源包中还有一个 WaiGuaDll.dll 文件，读者可以测试一下。

导入表注入的优点和缺点如下。

- 优点：游戏进程常会导入一些系统或第三方的库，这些库数量多、类型杂，且不易用完整性校验来防御这种修改导入表的行为。因为每个客户端的库版本可能不同，所以这种加载方式比较隐蔽。又因为加载时间早于程序运行的时间，所以可以在加载之后恢复之前备份的 PE 文件，从而躲避检测。
- 缺点：文件操作比较明显，比较容易被分析人员使用 ProcessMonitor 等工具检测到。如果能配合第三方进程加载一起使用，效果会更好。

2.7 劫持进程创建注入

如果能在目标进程运行起来之前获取目标进程的读、写等权限，那么注入将容易得多。为了实现这个目标，让我们先研究一下进程创建的关键 API 和调用序列。

Windows 下创建进程的 API 是 CreateProcess，它的原型如下。

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes。
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

这里不再赘述各个参数的具体含义，读者可以参考 MSDN 网站上的介绍。下面主要介绍 dwCreationFlags 的一个值—— CREATE_SUSPENDED 的含义。

当我们给参数 dwCreationFlags 赋值 CREATE_SUSPENDED 来调用 CreateProcess() 函数的时候，该进程的主线程将以悬挂方式启动。这个时候，要想恢复主线程的运行，需要调用 ResumeThread() 函数。劫持进程加载就是在主线程被悬挂和恢复的这段时间内，把 PE 或代码注入目标进程的。

劫持进程的创建过程如图 2-17 所示。

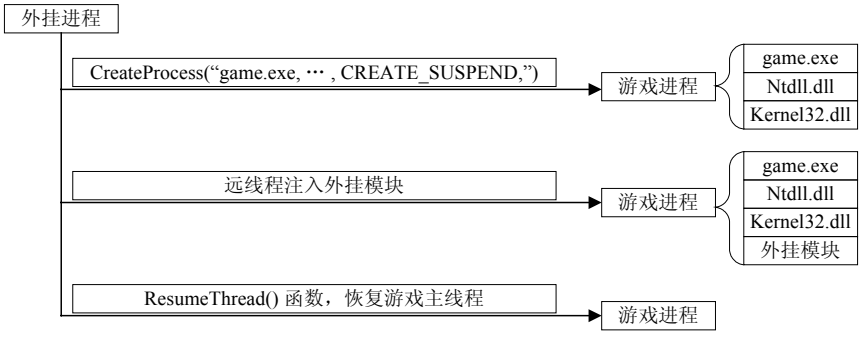


图 2-17　劫持进程的创建过程

图 2-17 中的过程可以总结为以下 3 步。

（1）调用以悬挂方式调用 CreateProcess() 函数，创建游戏进程。

（2）采用远线程方式注入外挂模块。

（3）调用 ResumThread() 函数恢复游戏主线程。

这种加载方式比较简单，这里就不给出演示程序和代码了，读者可以自己实践一下。

由于现在很多游戏进程在启动时采用双进程保护机制，所以即使通过上面的劫持进程成功实现加载，也只不过是注入双进程保护的第一个进程。为了再次使用劫持进程创建机制来注入真正的游戏进程，让我们一起认识一下用户态下进程创建的函数调用序列，如图 2-18 所示。

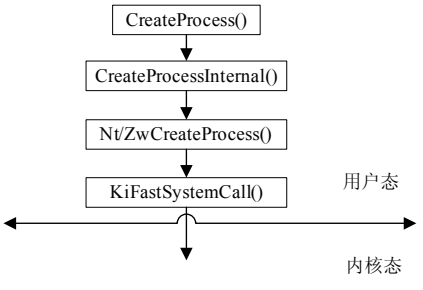


图 2-18　用户态进程创建调用序列

在图 2-18 中，读者不怎么熟悉的可能就是 KiFastSystemCall() 函数了。这个函数

相当于 ring3 转 ring0 的一个通道。CreateProcess() 函数最后通过 KiFastSystemCall() 函数进入内核态。我们可以在任意层次上通过 Hook 对应的函数来创建劫持进程，以达到加载的目的。例如，在 ring3 即将进入 ring0 之际，即调用 KiFastSystemCall() 函数的时候，劫持这个函数的调用，可以实现加载。

劫持进程创建注入的优点和缺点如下。

- 优点：不易防御，成功率较高。
- 缺点：劫持时间不宜过长，否则会被发现；劫持后，如需再次劫持，则需要 Hook，因此易被检测。

2.8 LSP 劫持注入

LSP（Layer Service Provider，分层服务提供者）是一个 DLL 程序，安装在 winsock 目录中，依靠底层的基础服务提供者来实现高层的服务。

LSP 的体系结构如图 2-19 所示。

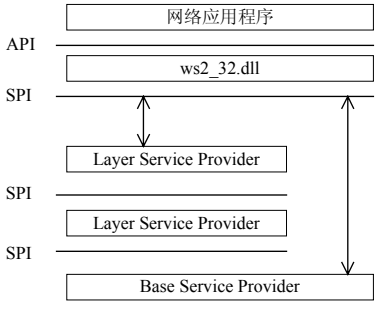


图 2-19 LSP 的体系结构

SPI（Service Provider Interface，服务提供者接口）是由分层服务提供者导出的供 ws2_32.dll 调用的系列函数。例如，与 winsock API 对应的 SPI 函数有 WSPStartup、WSPSocket、WSPSendTo 等。当网络应用程序调用 WSASocket/socket 函数创建套接字时，会有 3 个参数，分别是地址族、套接字类型和协议。正是这 3 个参数共同决定了由哪种传输服务提供者（基础服务提供者、分层服务提供者）来实现本应用程序的功能，这样便实现了 LSP 的加载功能。

SPI 提供的 3 种协议是分层协议、基础协议和协议链。分层协议在基础协议的上

层，依靠底层基础协议实现高级的通信服务。基础协议是能够独立、安全地与远程端点实现数据通信的协议，它是相对于分层协议而言的。协议链是将一系列的基础协议和分层协议按特定的顺序连接在一起的链状结构。

Winsock 目录用 WSAPROTOCOL_INFO 结构描述特定协议的完整信息，一个 WSAPROTOCOL_INFO 结构称为一个 winsock 目录入口，协议类型由 WSAPROTOCOL_INFOW 结构内 WSAPROTOCOLCHAIN 结构中的数据指定，示例如下。

```
typedef struct _WSAPROTOCOLCHAIN {
// 链的大小，也就是下面数组的大小
int ChainLen;
// 协议链入口数组，数组成员为链中协议的目录 ID
DWORD ChainEntries[MAX_PROTOCOL_CHAIN];
} WSAPROTOCOLCHAIN, *LPWSAPROTOCOLCHAIN;
```

ChainLen 暗示了入口的提供者类型："0" 代表分层协议；"1" 代表基础协议（基础提供者，如 TCP 和 UDP 提供者，由与之关联的内核模式协议驱动 TCPIP.SYS）；"2" 代表协议链。

图 2–20 描述了一个叫做 "外挂 LSP" 的分层协议通过一个叫做 "外挂 LSP over [TCP/IP]" 的协议链挂接在微软的 TCP 提供者上的例子。

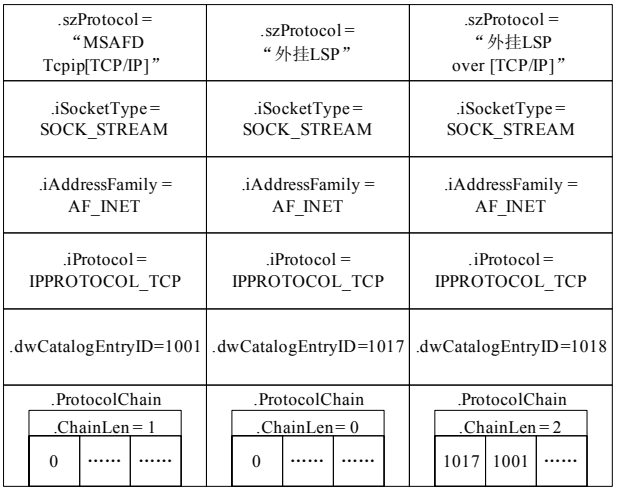


图 2-20　在微软 TCP 提供者上挂接 LSP 的例子

根据上面对 LSP 的分析可知，要实现 LSP 加载，大致有以下两个步骤。

（1）实现 LSP，即编写一个导出 WSPStartup() 函数的 DLL 文件。

（2）安装该 LSP。

下面就让我们看看这两个步骤的具体实现。

2.8.1 编写 LSP

winsock2 LSP 是一个标准的 Windows DLL，而且，每个 LSP 必须实现和导出 WSPStartup() 函数。WSPStartup() 函数的原型如下。

```
    int WSPStartup(
// 调用者可以使用的 winsock SPI 的最高版本号，高字节是低版本号
WORD wVersionRequested,
// 指向一个 WSPDATA 结构，用于取得 winsock 服务提供者的详细信息
LPWSPDATA lpWSPData,
// 指向一个 WSAPROTOCOL_INFO 结构，用来指定想得到的协议特征
LPWSAPROTOCOL_INFO lpProtocolInfo,
    // Ws2_32.dll 提供的向上调用转发的函数表结构
    WSPUPCALLTABLE UpcallTable,
// 指向 SPI 函数表结构的指针，用于返回 30 个 SPI 服务函数
LPWSPPROC_TABLE lpProcTable
);
```

WSPStartup 的处理流程大致如图 2–21 所示。WSPStartup() 函数内部的实现流程大致如下。

（1）winsock 应用程序调用 WSASocket/socket 后，系统根据协议链加载顶层的目录 ID 为 1017 的 LSP，即外挂 LSP。

（2）调用外挂 LSP 中导出的 WSPStartup() 函数，lpProtocolInfo 参数根据接收到的 WSAPROTOCOL_INFO 结构体找到下层提供者的目录 ID（1001），再枚举所有提供者，找到下层提供者入口的 WSAPROTOCOL_INFOW 结构。

（3）使用 WSCGetProviderPath() 函数得到下层提供者的 DLL 路径，即防火墙 LSP 的路径。

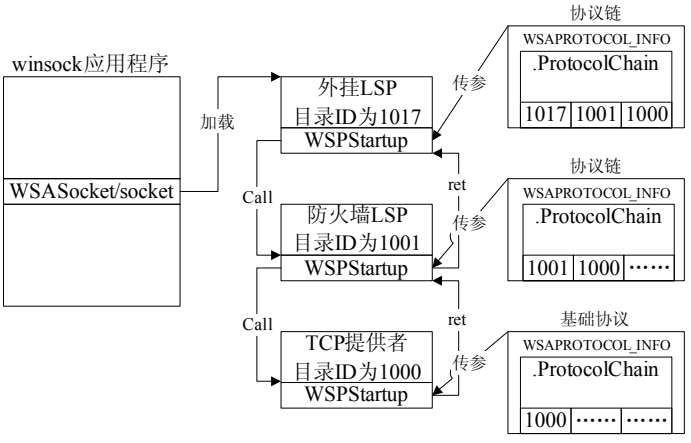


图 2-21　WSPStartup 递归执行流程

（4）由于 DLL 路径可能包含环境变量，所以使用 ExpandEnvironmentStrings() 函数将它展开。

（5）用 LoadLibrary 加载该 DLL，调用 GetProcAddress，取得 WSPStartup() 函数的指针。

（6）调用该 WSPStartup() 函数继续初始化下层提供者。

WSPStartup() 函数的具体的代码如下，读者也可以参考本章资源包中 LSPInject 工程的代码。

```
int WSPAPI WSPStartup(
  WORD wVersionRequested,
  LPWSPDATA lpWSPData,
  LPWSAPROTOCOL_INFO lpProtocolInfo,
  WSPUPCALLTABLE UpcallTable,
  LPWSPPROC_TABLE lpProcTable
)
{

    // 加载外挂模块
    if(lpProtocolInfo->ProtocolChain.ChainLen <= 1)
    {
        return WSAEPROVIDERFAILEDINIT;
```

```
        }

        // 保存向上调用的函数表指针（这里我们不使用它）
        g_pUpCallTable = UpcallTable;

        // 枚举协议，找到下层协议的 WSAPROTOCOL_INFOW 结构
        WSAPROTOCOL_INFOW   NextProtocolInfo;
        int nTotalProtos;
        LPWSAPROTOCOL_INFOW pProtoInfo = GetProvider(&nTotalProtos);
        // 下层入口 ID
        DWORD dwBaseEntryId=lpProtocolInfo->ProtocolChain.ChainEntries[1];
        for(int i=0; i<nTotalProtos; i++)
        {
            if(pProtoInfo[i].dwCatalogEntryId == dwBaseEntryId)
            {
                memcpy(&NextProtocolInfo, &pProtoInfo[i], sizeof(Next
ProtocolInfo));
                break;
            }
        }
        if(i >= nTotalProtos)
        {
            ODS(L" WSPStartup: Can not find underlying protocol \n");
            return WSAEPROVIDERFAILEDINIT;
        }

        // 加载下层协议的 DLL
        int nError;
        TCHAR szBaseProviderDll[MAX_PATH];
        int nLen = MAX_PATH;
        // 取得下层提供程序的 DLL 路径
        if(::WSCGetProviderPath(&NextProtocolInfo.ProviderId,   szBase
ProviderDll, &nLen, &nError) == SOCKET_ERROR)
        {
```

```
            return WSAEPROVIDERFAILEDINIT;
        }
        if(!::ExpandEnvironmentStrings(szBaseProviderDll,
szBaseProviderDll, MAX_PATH))
        {
            return WSAEPROVIDERFAILEDINIT;
        }
        // 加载下层提供程序
        HMODULE hModule = ::LoadLibrary(szBaseProviderDll);
        if(hModule == NULL)
        {
            return WSAEPROVIDERFAILEDINIT;
        }

        // 导入下层提供程序的 WSPStartup()函数
        LPWSPSTARTUP  pfnWSPStartup = NULL;
        pfnWSPStartup    =    (LPWSPSTARTUP)::GetProcAddress(hModule,
"WSPStartup");
        if(pfnWSPStartup == NULL)
        {
            return WSAEPROVIDERFAILEDINIT;
        }

        // 调用下层提供程序的 WSPStartup()函数
        LPWSAPROTOCOL_INFOW pInfo = lpProtocolInfo;
        if(NextProtocolInfo.ProtocolChain.ChainLen == BASE_PROTOCOL)
            pInfo = &NextProtocolInfo;

        int nRet = pfnWSPStartup(wVersionRequested, lpWSPData, pInfo,
UpcallTable, lpProcTable);
        if(nRet != ERROR_SUCCESS)
        {
            return nRet;
        }
```

```
        // 保存下层提供者的函数表
        g_NextProcTable = *lpProcTable;

        // 修改传递给上层的函数表，Hook 感兴趣的函数
        // 这里作为示例，仅 Hook 了 WSPSendTo()函数
        // 读者还可以 Hook 其他函数，如 WSPSocket、WSPCloseSocket、WSPConnect 等
        // lpProcTable->lpWSPSendTo = WSPSendTo;
        // lpProcTable->lpWSPSend  = WSPSend;

        FreeProvider(pProtoInfo);
        return nRet;
    }
```

2.8.2 安装 LSP

事实上，安装 LSP 就是安装一个分层协议和一个描述这个分层协议挂接信息的协议链，具体如下。分层协议和协议链都是由 WSAPROTOCOL_INFO 这个结构体来描述的。

（1）安装分层协议。通过 WSCEnumProtocols 函数枚举所有的服务提供者，然后根据地址族和协议类型（如 AF_INET 和 IPPROTO_TCP）找到需要安装在其上的一个服务提供者，修改这个服务提供者（WSAPROTOCO_INFOW）的一些成员，然后通过 WSCInstallProvider() 函数安装该分层协议。

（2）获取上一步中安装的分层协议的目录 ID，安装协议链。这一步操作涉及一个可能会移动数组成员的操作。当下层服务提供者是协议链的时候，其 ChainEntries 中的成员首先需要向后移动，然后把上一步中安装的分层协议的目录 ID 放入 ChainEntries[0] 中。

（3）枚举所有协议，重排 winsock 目录。重排 winsock 目录，就是把要安装的协议链的目录 ID 放在一个数组的首位，把其他协议链的目录 ID 放在其后，然后通过 WSCWriteProviderOrder() 函数重排所有协议。

因为安装 LSP 的步骤比较固定且简单，这里就不详细描述了，读者可以参考本章资源包中的 InstDemo 工程，其核心代码在 InstallProvider 函数中，具体如下。

```
    // 将 LSP 安装到 TCP 协议提供者之上
    int InstallProvider(WCHAR *wszDllPath)
    {
       WCHAR wszLSPName[] = L"外挂 LSP";    // LSP 的名称
       int nError = NO_ERROR;
       LPWSAPROTOCOL_INFOW pProtoInfo;
       int nProtocols;
       // 要安装的 TCP 分层协议和协议链
       WSAPROTOCOL_INFOW TCPLayeredInfo, TCPChainInfo;
    DWORD dwTCPOrigCatalogId, dwLayeredCatalogId;

       // 在 winsock 目录中找到原来的 TCP 协议服务提供者，LSP 要安装在它之上
       // 枚举所有服务程序提供者
       // 注意：下面是安装到 TCP 上的
       pProtoInfo = GetProvider(&nProtocols);
       for(int i=0; i<nProtocols; i++)
       {
           if(pProtoInfo[i].iAddressFamily == AF_INET &&
     pProtoInfo[i].iProtocol == IPPROTO_TCP)
           {
               memcpy(&TCPChainInfo,                    &pProtoInfo[i],
sizeof(TCPLayeredInfo));
       TCPChainInfo.dwServiceFlags1 = TCPChainInfo.dwServiceFlags1 &
    ~XP1_IFS_HANDLES;
               // 保存原来的入口 ID
               dwTCPOrigCatalogId = pProtoInfo[i].dwCatalogEntryId;
               break;
           }
       }

       // 安装分层协议，获取一个 winsock 库安排的目录 ID，即 dwLayeredCatalogId
       // 直接使用下层协议的 WSAPROTOCOL_INFOW 结构
       memcpy(&TCPLayeredInfo, &TCPChainInfo, sizeof(TCPLayeredInfo));
```

```
        // 修改协议的名称和类型，设置 PFL_HIDDEN 标志
        wcscpy(TCPLayeredInfo.szProtocol, wszLSPName);
    // LAYERED_PROTOCOL 即 0
        TCPLayeredInfo.ProtocolChain.ChainLen = LAYERED_PROTOCOL;
    TCPLayeredInfo.dwProviderFlags |= PFL_HIDDEN;
        // 安装
        if(::WSCInstallProvider(&ProviderGuid,
                        wszDllPath,  &TCPLayeredInfo,  1,  &nError)  ==
SOCKET_ERROR)
            return nError;
        // 重新枚举协议，获取分层协议的目录 ID
        FreeProvider(pProtoInfo);
        pProtoInfo = GetProvider(&nProtocols);
        for(i=0; i<nProtocols; i++)
        {
            if(memcmp(&pProtoInfo[i].ProviderId, &ProviderGuid, sizeof
(ProviderGuid)) == 0)
            {
                dwLayeredCatalogId = pProtoInfo[i].dwCatalogEntryId;
                break;
            }
        }
        // 安装协议链
        // 修改协议的名称和类型
        WCHAR wszChainName[WSAPROTOCOL_LEN + 1];
        swprintf(wszChainName,    L"%ws    over    %ws",    wszLSPName,
TCPChainInfo.szProtocol);
        wcscpy(TCPChainInfo.szProtocol, wszChainName);
        if(TCPChainInfo.ProtocolChain.ChainLen == 1)
        {
            TCPChainInfo.ProtocolChain.ChainEntries[1] = dwTCPOrigCatalogId;
        }
        else
        {
            for(i=TCPChainInfo.ProtocolChain.ChainLen; i>0 ; i--)
```

```
            {
    TCPChainInfo.ProtocolChain.ChainEntries[i] = TCPChainInfo.Protocol
Chain.ChainEntries[i-1];
            }
        }
        TCPChainInfo.ProtocolChain.ChainLen ++;
        // 将分层协议置于此协议链的顶层
        TCPChainInfo.ProtocolChain.ChainEntries[0] = dwLayeredCatalogId;
        // 获取一个 Guid 并安装它
        GUID ProviderChainGuid;
        if(::UuidCreate(&ProviderChainGuid) == RPC_S_OK)
        {
            if(::WSCInstallProvider(&ProviderChainGuid,
                    wszDllPath, &TCPChainInfo, 1, &nError) == SOCKET_
ERROR)
                    return nError;
        }
        else
            return GetLastError();

        // 重新排序 winsock 目录，将刚刚安装的协议链提前
        // 重新枚举安装的协议
        FreeProvider(pProtoInfo);
        pProtoInfo = GetProvider(&nProtocols);
        DWORD dwIds[20];
        int nIndex = 0;
        // 添加刚刚安装的协议链
        for(i=0; i<nProtocols; i++)
        {
            if((pProtoInfo[i].ProtocolChain.ChainLen > 1) &&
            (pProtoInfo[i].ProtocolChain.ChainEntries[0] == dwLayered
CatalogId))
                dwIds[nIndex++] = pProtoInfo[i].dwCatalogEntryId;
        }
        // 添加其他协议
```

```
        for(i=0; i<nProtocols; i++)
        {
            if((pProtoInfo[i].ProtocolChain.ChainLen <= 1) ||
            (pProtoInfo[i].ProtocolChain.ChainEntries[0] != dwLayered
CatalogId))
                dwIds[nIndex++] = pProtoInfo[i].dwCatalogEntryId;
        }
        // 重新排序 winsock 目录
        nError = ::WSCWriteProviderOrder(dwIds, nIndex);
        FreeProvider(pProtoInfo);
        return nError;
    }
```

2.9 输入法注入

输入法注入是指在自制的输入法加载到目标进程的时候，附带加载真正待注入DLL 的一种注入方式。

要想在 Windows 平台上实现输入法注入，必须了解一些与 Windows 输入法相关的知识，具体细节读者可以上网搜索，下面仅列出与本节注入主题相关的内容。

- 根据 Windows 的规定，输入法就是一个 DLL，只不过它是一个特殊的 DLL，必须具有标准输入法程序所规定的那些接口。
- 输入法在系统目录中是以“ime”为扩展名的文件。当在应用程序中激活某个输入法时，输入法管理器（imm32.dll）就会在该应用程序的进程中加载对应的 IME 文件。加载 IME 文件与加载普通的 DLL 文件没有本质区别，所以可以认为，输入法就是注入应用程序的一个 DLL 文件。
- 实现输入法注入以后，可以在窗口中切换到其他输入法，这不会影响已经注入的 DLL。

输入法注入的实现，一般需要 3 个程序协同完成：一个是自制的输入法模块，即以“ime”结尾的文件；一个是真正待加载的 DLL 模块；一个是控制 IME 文件安装的 EXE 模块。本章资源包中的 ImeInjectControl 工程给出了在 Visual C++ 6.0 下实现的这 3 个模块，其名称分别是 ImeDllLoader.ime、InjectDll.dll 和 ImeInject.exe，详细介绍如下。

- ImeDllLoader.ime 是根据网上的一份代码改编的，其原始代码采用 DLL 共享内存的方式来传递参数，在这里则采用命名管道的方式来传递参数。
- ImeInject.exe 的核心是调用 ImmInstall IME 来安装 ImeDllLoader.ime 输入法。ImmInstallIME 函数的声明如下。

```
HKL ImmInstallIME(LPCTSTR lpszIMEFileName, LPCTSTR lpszLayoutText);
```

该函数的两个参数分别为输入法 IME 文件的文件名和输入法名，调用该函数后将返回一个所安装输入法的标识符（或称输入法句柄），具体用法如下。

```
// 安装 IME 输入法文件
HKL hImeFile = ImmInstallIME(strSystemDir, "外挂输入法");
if( ImmIsIME(hImeFile) )
{
    // 安装成功
    bSuccess = true;
    printf("输入法安装成功");
}
else
{
    // 安装失败
    // MessageBox(NULL, "安装失败", "安装输入法", MB_OKCANCEL);

    printf("输入法安装失败 %d",GetLastError());
}
```

- InjectDll.dll 是一个被 ImeDllLoader.ime 加载到目标进程空间的 DLL Demo。

输入法加载的模型如图 2-22 所示，流程如下。

（1）ImeInject.exe 创建 PipeServerThread 线程，该线程调用 PipeServer() 函数创建命名管道的服务端，等待命名管道的客户端的连接，示例如下。

```
// 创建管道服务器线程 PipeServerThread
HANDLE  hNewThread = CreateThread(NULL, NULL,
(LPTHREAD_START_ROUTINE)PipeServerThread, NULL, NULL, NULL);
if ( !hNewThread )
{
```

```
        OutputDebugStr("[!] SetupIME CreateThread error\n");
            return FALSE;
    }
// PipeServer 创建命名服务端，等待客户端连接
DWORD WINAPI PipeServerThread(
                        LPVOID lpThreadID
                    )
{

    PipeServer("\\\\.\\pipe\\winsun_ime_load_dll",g_hEvent);
    return 1;
}
```

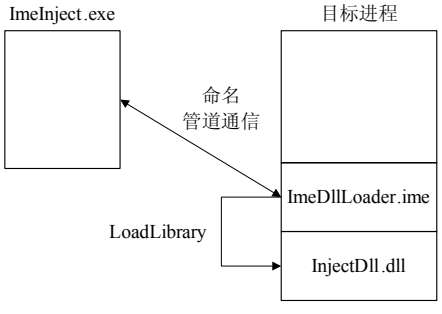


图 2-22 输入法加载模型

（2）ImeInject.exe 调用 ImmInstall IME 函数安装 ImeDllLoader.ime 输入法。

（3）打开目标进程（如 notepad.EXE），激活 ImeDllLoader.ime 输入法，则该输入法模块被系统加载到目标进程地址空间。

（4）ImeDllLoader.ime 加载到目标进程空间后，将创建 ImeDllLoadThread 线程。该线程调用 PipeClient() 函数创建命名管道的客户端，并连接命名管道的服务端以接收控制命令，示例如下。

```
// ImeDllLoadThread 线程调用 PipeClient() 函数
DWORD WINAPI ImeDllLoadThread(
                        LPVOID lpThreadID
                        )
```

```
{
    OutputDebugStr("[Winsun] ImeDllLoadThread Started! \n");
    PipeClient("\\\\.\\pipe\\winsun_ime_load_dll");
    return 1;
}
```

（5）PipeClient() 函数根据接收到的控制命令和参数进行加载或卸载 InjectDll.dll 的操作，示例如下。

```
// PipeClient()函数接收 PipeServer()函数的命令
void PipeClient(LPCSTR lpPipeName)
{
        HANDLE hPipeClient = 0;
        DWORD  dwReadBytes = 0;
        HMODULE hDll = NULL;
        PIPE_COMMUNICATION stPipeCom={0};
        // 等待 Pipe Server 的命令
        if( WaitNamedPipe(lpPipeName, NMPWAIT_WAIT_FOREVER) )
         {
             // 等待成功后，打开命名管道
             hPipeClient = CreateFile(
                 lpPipeName,
                 GENERIC_READ | GENERIC_WRITE,
                 FILE_SHARE_READ | FILE_SHARE_WRITE,
                 NULL,
                 OPEN_EXISTING,
                 FILE_ATTRIBUTE_ARCHIVE | FILE_FLAG_WRITE_THROUGH,
                 NULL
                 );
             if( INVALID_HANDLE_VALUE == hPipeClient )
             {
                OutputDebugStr("[XL] PipeClient CreateFile fail!\n");
                return ;
             }

             while(TRUE)
             {
```

```
                    // 从管道中读取命令
                    ReadFile(hPipeClient, &stPipeCom,
    sizeof(PIPE_COMMUNICATION), &dwReadBytes, NULL);
                    switch(stPipeCom.Cmd)
                    {
                        case 1:
                        // 加载
                        hDll = LoadLibrary(stPipeCom.DLLString);
                        break;

                        case 2:
                            // 卸载
                            FreeLibrary(hDll);
                        break;
                    }
                    Sleep(100);
                }

            }
    }
```

下面是输入法加载的操作步骤。

（1）将 ImeDllLoader.ime、InjectDll.dll 和 ImeInject.exe 放入同一目录，如图 2-23 所示。

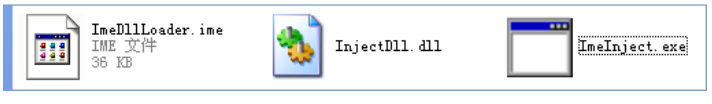


图 2-23　将 3 个程序放在同一目录下

（2）双击 ImeInject.exe 文件图标，打开如图 2-24 所示的命令行窗口。

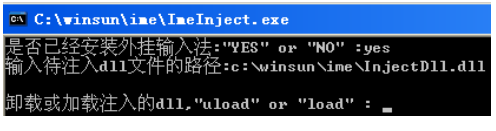


图 2-24　ImeInject.exe 命令行窗口

ImeInject 程序可以安装新的输入法或跳过已安装的输入法，并且需要提供待加载 DLL 的路径。当输入法安装成功后，可以通过接收 uload 或 load 命令来卸载或加载 InjectDll.dll。如图 2-25 所示，外挂输入法已经成功安装。当打开某个 GUI 程序并切换到外挂输入法的时候，该输入法模块就会加载到该 GUI 程序的进程空间。

如图 2-26 所示，在 notepad.exe 使用外挂输入法时，系统将 ImeDllLoader.ime 加载到 notepad.exe 地址空间。

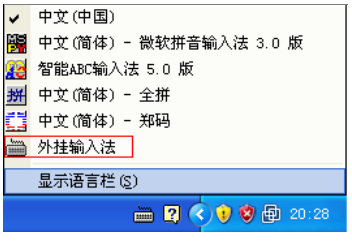


图 2-25　成功安装输入法

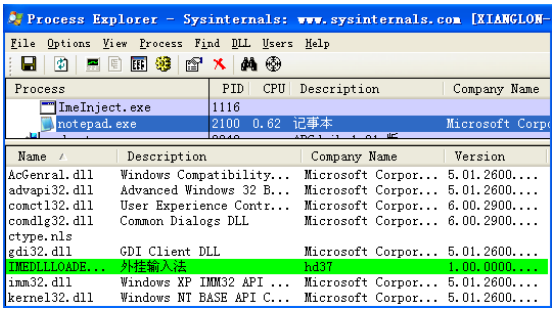


图 2-26　notepad.exe 进程中的模块

如图 2-27 所示，ImeInject.exe 程序收到 load 命令后，就可以使 ImeDllLoader.ime 调用 LoadLibrary 加载 InjectDll.dll 模块进入 notepad.exe 地址空间了。为了演示方便，在这里，InjectDll.dll 加载成功后会调用 MessageBox() 函数弹出对话框。同理，当 ImeInject.exe 收到 uload 命令后，ImeDllLoader.ime 就会调用 FreeLibrary() 函数卸载 notepad.exe 中的 InjectDll.dll 模块。

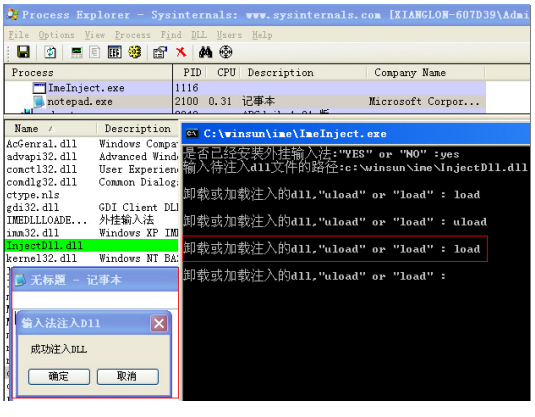


图 2-27　notepad.exe 成功加载 InjectDll.dll

输入法注入的优点和缺点如下。

- 优点：加载方法优美，难以阻止，实现简单，使用方便。
- 缺点：需要考虑输入法模块本身不被提取的特点，所以，最好在外挂模块加载到目标进程地址空间后快速隐藏或卸载输入法模块。

2.10 ComRes 注入

ComRes.dll 是 Windows 操作系统中 COM 服务所使用的一个系统文件，常被木马病毒替换用于注入。ComRes.dll 加载是指用一个假的 ComRes 模块覆盖原来的 ComRes 模块，从而实现注入。

用 IDA 对 ComRes.dll 模块进行分析之后，可以发现这个 DLL 非常简单，只导出一个 COMResModuleInstance() 函数负责返回模块句柄，而模块的入口 DllMain() 函数负责保存模块句柄。假的 ComRes.dll 代码见本章资源包中的 FakeComRes 工程。

ComRes 加载过程分以下 3 步。

（1）以 ComRes 模块的规范编写外挂模块。

（2）关闭文件保护。可以使用现成的工具，如 Windows XP 下的 XPLite，或者自己编写代码。

（3）将原始 ComRes 模块删除或重命名，然后把外挂模块（ComRes.dll）复制到系统目录下。

ComRes 注入的优点和缺点如下。

- 优点：实现简单，操作方便。
- 缺点：如果不进行模块隐藏，因带有明显特征而易被检测到。

第 3 章　浅谈无模块化

本章主要讨论模块注入目标进程后实现自我隐藏的方法。在安全领域，无论是木马、Rootkit，还是外挂，隐藏自己都是至关重要的技术。因为本书研究的外挂都是用户态的，所以这里的模块隐藏也指用户态的隐藏。

在用户态实现模块隐藏，虽然感觉上施展空间不大，而且技术含量不高，但是，不管“黑猫白猫”，能抗检测、抗分析的就是“好猫”。

比较常见的模块隐藏方法有抹去模块的 PE 头、断开进程的 LDR_MODULE 链、Hook 模块枚举的函数等。

接下来，就让我们研究一下模块隐藏的两种常见且好用的方法。

- 断开进程的 LDR_MODULE 链。
- 抹去模块的 PE 头。

3.1　LDR_MODULE 隐藏

在 Windows 操作系统中，每个被映射到进程地址空间的模块（EXE 或 DLL）都有唯一的 LDR_MODULE 结构体与之对应，这个结构体是 Windows 操作系统感知用户态模块存在的依据。

下面就让我们先了解一下 LDR_MODULE 的结构体声明，示例如下。

```
typedef struct _LDR_MODULE
{
  LIST_ENTRY        InLoadOrderModuleList;
  LIST_ENTRY        InMemoryOrderModuleList;
  LIST_ENTRY        InInitializationOrderModuleList;
  void*             BaseAddress;
  void*             EntryPoint;
  ULONG             SizeOfImage;
  UNICODE_STRING    FullDllName;
  UNICODE_STRING    BaseDllName;
  ULONG             Flags;
  SHORT             LoadCount;
  SHORT             TlsIndex;
  HANDLE            SectionHandle;
  ULONG             CheckSum;
  ULONG             TimeDateStamp;
} LDR_MODULE, *PLDR_MODULE;
```

从上面的结构体声明中可以得到很多与模块相关的重要信息，举例如下。

- InLoadOrderModuleList：代表按加载顺序构成的模块链表。
- InMemoryOrderModuleList：代表按内存顺序构成的模块链表。
- InInitializationOrderModuleList：代表按初始化顺序构成的模块链表。
- BaseAddress：代表该模块的基地址。
- EntryPoint：代表该模块的入口。
- SizeOfImage：代表该模块的映像大小。
- FullDllName：代表包含路径的模块名。
- BaseDllName：代表不包含路径的模块名。
- LoadCount：代表该模块的引用计数。

这里有 3 个字段特别重要，分别是 InLoadOrderModuleList、InMemoryOrderModuleList 和 InInitializationOrderModuleList，它们分别指明了该模块所处的链表以及该模块加载到进程地址空间的方式。

对于 Windows 提供的枚举模块的 API，如 Module32First 和 Module32Next，都会

遍历这个 LDR_MODULE 链表。我们只要把模块所对应的 LDR_MODULE 结构从这 3 个链表上断开，就可以隐藏模块。

下面让我们看看如何通过调试工具 Windbg 从 FS:[30] 定位到 LDR_MODULE 链表。

（1）输入命令“dd fs:[30]”，定位 PEB（进程环境块）。如图 3-1 所示，PEB 的地址是 7ffde000。

```
0:001> dd fs:[30]
0038:00000030  7ffde000 00000000 00000000 00000000
0038:00000040  00000000 00000000 00000000 00000000
0038:00000050  00000000 00000000 00000000 00000000
0038:00000060  00000000 00000000 00000000 00000000
0038:00000070  00000000 00000000 00000000 00000000
0038:00000080  00000000 00000000 00000000 00000000
0038:00000090  00000000 00000000 00000000 00000000
0038:000000a0  00000000 00000000 00000000 00000000
```

图 3-1　定位 PEB

（2）输入命令“dt _PEB 7ffde000”，定位 PEB_LDR_DATA。如图 3-2 所示，用方框标出的部分表明 PEB_LDR_DATA 的地址为 0x00251e90。

```
0:001> dt _PEB 7ffde000
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged     : 0x1 ''
   +0x003 SpareBool         : 0 ''
   +0x004 Mutant            : 0xffffffff
   +0x008 ImageBaseAddress  : 0x4ad00000
   +0x00c Ldr               : 0x00251e90 _PEB_LDR_DATA
   +0x010 ProcessParameters : 0x00020000 _RTL_USER_PROCESS_PARAMETERS
```

图 3-2　定位 PEB_LDR_DATA

（3）输入命令“dt _PEB_LDR_DATA 00251e90”，定位 LDR_MODULE。如图 3-3 所示，方框标出的部分就是 LDR_MODULE 链的头节点。

```
0:001> dt _PEB_LDR_DATA 0x00251e90
   +0x000 Length           : 0x28
   +0x004 Initialized      : 0x1 ''
   +0x008 SsHandle         : (null)
   +0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x251ec0 - 0x252e40 ]
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x251ec8 - 0x252e48 ]
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x251f28 - 0x252e50 ]
   +0x024 EntryInProgress  : (null)
```

图 3-3　定位 LDR_MODULE 链

（4）输入命令“dt _ldr_data_table_entry 0x251ec0”，输出 cmd 进程模块的结构。如图 3-4 所示，windbg 中的_LDR_DATA_TABLE_ENTRY 就是 LDR_MODULE 符号。

```
0:001> dt _ldr_data_table_entry 0x251ec0
ntdll!LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x251f18 - 0x251e9c ]
   +0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x251f20 - 0x251ea4 ]
   +0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x018 DllBase          : 0x4ad00000
   +0x01c EntryPoint       : 0x4ad05046
   +0x020 SizeOfImage      : 0x75000
   +0x024 FullDllName      : _UNICODE_STRING "C:\WINDOWS\system32\cmd.exe"
   +0x02c BaseDllName      : _UNICODE_STRING "cmd.exe"
   +0x034 Flags            : 0x5000
   +0x038 LoadCount        : 0xffff
   +0x03a TlsIndex         : 0
   +0x03c HashLinks        : _LIST_ENTRY [ 0x252dd4 - 0x7c99e270 ]
   +0x03c SectionPointer   : 0x00252dd4
   +0x040 CheckSum         : 0x7c99e270
   +0x044 TimeDateStamp    : 0x48025baf
   +0x044 LoadedImports    : 0x48025baf
   +0x048 EntryPointActivationContext : (null)
   +0x04c PatchInformation : (null)
```

图 3-4　cmd.exe 模块的 LDR_MODULE 结构体

对以上搜索 LDR_MODULE 链表的过程进行逻辑上的总结，如图 3–5 所示。

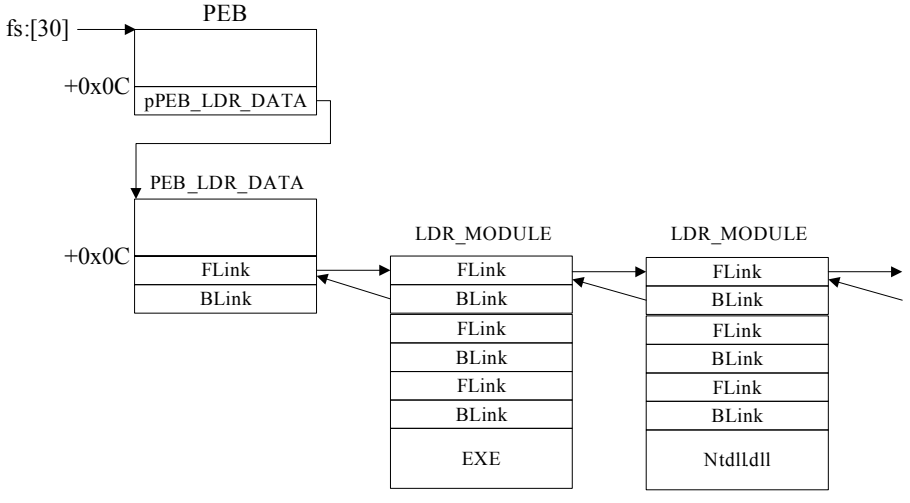


图 3-5　LDR_MODULE 链表

在图 3–5 中，InMemoryOrderModuleList 和 InInitializationOrderModuleList 两条链表的结构没有给出，只给出了 InLoadOrderModuleList 链表的结构。只要断开图 3–5 中的 3 条链表，就能使 Windows 操作系统提供的 API 无法感知进程中的这个模块。断开 LDR_MODULE 链表后的情况如图 3–6 所示。

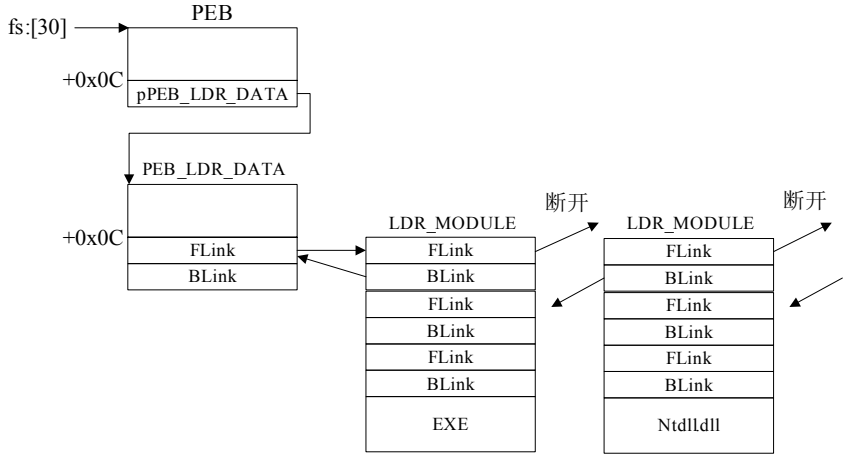


图 3-6　断开 LDR_MODULE 链表

下面我们看看如何从代码的角度来断开 LDR_MODULE 链。

```
// 方法一：伪代码
typedef struct _LDR_MODULE
  {
      LIST_ENTRY InLoadOrderModuleList;
      LIST_ENTRY InMemoryOrderModuleList;
      LIST_ENTRY InInitializationOrderModuleList;
      PVOID BaseAddress;
      PVOID EntryPoint;
      ULONG SizeOfImage;
      UNICODE_STRING FullDllName;
      UNICODE_STRING BaseDllName;
      ULONG Flags;
      USHORT LoadCount;
      USHORT TlsIndex;
      LIST_ENTRY HashTableEntry;
      ULONG TimeDateStamp;
 } LDR_MODULE, *PLDR_MODULE, *PLML;
typedef struct _LE{
  DWORD Flink;
```

```
  DWORD Blink;
}LE, LIST_ENTRY;
// 断开 LDR_MODULE 隐藏模块代码
BOOL BreakLdrModuleLink(DWORD dwBaseAddr)
{
  PLDR_MODULE pLMFNode = NULL, pLNode = NULL ;
  PLDR_MODULE pLMHNode = NULL, pLMPNode = NULL;
  PLDR_MODULE pLMTNode = NULL;
  BOOL bSuccess = FALSE;
// 获取 LDR_MODULE 链的头指针
__asm{
 pushad  // 保存通用寄存器
 pushfd  // 保存标志寄存器
 xor edx, edx
 mov ebx, fs:[edx + 0x30]  // 获取图 3-6 中的 PEB 地址
 mov ecx, [ebx + 0x0C]     // 获取图 3-6 中的 PEB_LDR_DATA 地址
 lea  edx, [ecx + 0x0C]
 mov ecx, [ecx + 0x0C]
 mov pLMHNode, edx
 mov pLMFNode, ecx         // 获取图 3-6 中的 LDR_MODULE 链首节点地址
 popfd  // 恢复标志寄存器
 popad  // 恢复通用寄存器
}
    // 查找目标
    PLDR_MODULE pLMNode = pLMFNode;
    pLMPNode = pLMHNode;
    do{
        // 判断是否为目标模块
        if( (DWORD)pLMNode->BaseAddress == dwBaseAddr)
        {
            bSuccess = TRUE;
            break;
        }
        pLMPNode = pLMNode;
        pLMNode = (PLDR_MODULE)pLMNode->InLoadOrderModuleList.Flink;
```

```
        }while(pLMNode != pLMHNode);

        if( !bSuccess )
        {
            OutputDebugString("cannot find the dest module!");
            return bSuccess;    // 未找到目标模块
        }

        // 断开 InLoadOrderModuleList 链
        // 重建 Flink
        pLMTNode = (PLDR_MODULE)pLMNode->InLoadOrderModuleList.Flink;
        pLMPNode->InLoadOrderModuleList.Flink = (PLIST_ENTRY)pLMTNode;
        // 重建 Blink
        ((PLDR_MODULE)(pLMNode->InLoadOrderModuleList.Flink))->
    InLoadOrderModuleList.Blink                                    =
pLMNode->InLoadOrderModuleList.Blink;

       // 断开 InMemoryOrderModuleList 链
       // 重建 Flink
        pLMPNode->InMemoryOrderModuleList.Flink =
        pLMNode->InMemoryOrderModuleList.Flink;
    // 重建 Blink
    pLMTNode = (PLML)(pLMNode->InMemoryOrderModuleList.Flink - sizeof
(LIST_ENTRY));
        pLMTNode->InMemoryOrderModuleList.Blink =
        pLMNode->InMemoryOrderModuleList.Blink;

       // 断开 InInitializationOrderModuleList 链
       // 重建 Flink
        pLMPNode->InInitializationOrderModuleList.Flink =
        pLMNode->InInitializationOrderModuleList.Flink;

    // 重建 Blink
    pLMTNode = (PLML)(pLMNode->InInitializationOrderModuleList.Flink -
2*sizeof(LIST_ENTRY));
        pLMTNode->InInitializationOrderModuleList.Blink   =  pLMNode->
InInitializationOrderModuleList.Blink;}
```

经过上面的努力，虽然我们已经可以使 Toolhelp、psapi 等枚举模块的 API 无法枚举目标模块，但是目前很多反外挂系统和逆向分析人员仍然可以通过搜索进程地址空间来找到 PE 模块。如果内存空间中有 n 个 PE 模块，但 LDR_MODULE 链上只有 $n-1$ 个 PE 模块，那么说明有 1 个 PE 模块从 LDR_MODULE 链上断开了。所以，为了进一步达到隐藏模块的目的，我们需要做得更加彻底——当模块注入目标进程空间后，首先断开 LDR_MODULE 链，然后抹去该模块的 PE“指纹”。第 3.2 节就将简单介绍抹去 PE 头的方法。

3.2　抹去 PE“指纹”

在 Win32 平台上（包括 Windows 9x/NT/2000/XP/2003/Vista/CE），EXE 文件、DLL 文件和 SYS 文件都是 PE 格式（Portable Executable File Format，可移植、可执行的文件格式）的。PE 文件被映射到内存后的基本结构如图 3-7 所示。要想抹去内存中的 PE 头，只要把“MZ”和“PE”标志去掉即可—— 这些标志就是 PE 格式文件区别于其他文件的“指纹”。

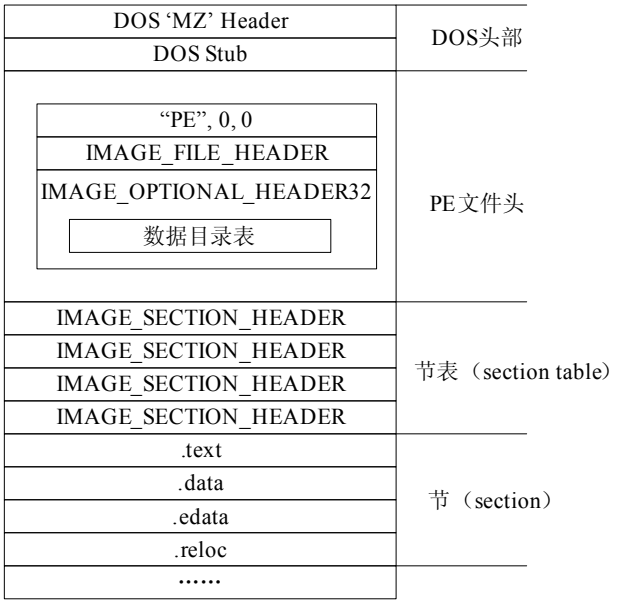


图 3-7　PE 文件格式的基本结构

下面让我们来看看如何从代码的角度抹去 PE 头中的“MZ”和“PE”标志。

```
// 伪代码
// 获取外挂模块的加载基地址
hGuaModule = GetModuleHandle("Gua.DLL");
// 改变 PE 头的页面保护
VirtualProtect((LPVOID) hGuaModule,1024,PAGE_READWRITE, &dwOldProtect);
// 开始定位 PE 头
PIMAGE_DOS_HEADER pDosHeader = (PIMAGE_DOS_HEADER) hGuaModule;
// 抹去“MZ”标志
pDosHeader-> e_magic = 0;
// DOS 头后面就是 PE 头
PIMAGE_NT_HEADERS pNtHeader = (PIMAGE_NT_HEADERS)(pDosHeader+1);
// 抹去“PE”标志
pNtHeader-> Signature = 0;

// 恢复 PE 头的原始页面保护
VirtualProtect((LPVOID) hGuaModule,1024,dwOldProtect, &dwOldProtect);
```

结合上面描述的断开 LDR_MODULE 链和抹去 PE 头的方法，本章给出了具体的示例程序 HideModule.exe，如图 3-8 所示，读者可以在本章资源包的 HideModule 工程中找到它。

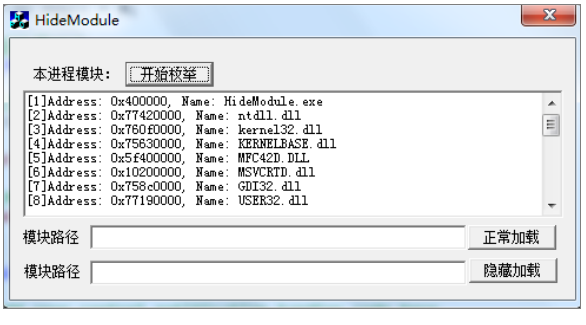


图 3-8　HideModule 示例程序

读者可以体验一下使用 HideModule 示例程序正常加载模块和隐藏加载模块的区别。正常加载某个模块之后，单击“开始枚举”按钮可以获取刚刚加载的模块信息。

但是，如果对某个模块单击“隐藏加载”按钮，那么单击“开始枚举”按钮则无法获取刚刚加载的模块信息。

3.3 本章小结

本章主要讨论在用户态下隐藏进程模块的实用方法，包括断开 LDR_MODULE 链和抹去 PE 头。虽然本章提及的方法比较简单，但是对 ring3 下的模块隐藏还是有不错的效果。当然，隐藏的方法还有很多，希望本章能起到抛砖引玉的作用。

第 4 章　安全的交互通道

从本质上看，外挂也是一种软件，也需要与用户进行交互，从而接收用户的操作指令来实现特定的功能，例如，按【F1】键是“加速”，按【F2】键是“无敌”，按【F3】键是“透视”等。虽然从程序的角度来实现这种交互并不难，但一款外挂要做到优秀，其中交互这一部分的关键还是要能很好地隐藏核心代码，而这才是难点。本章主要讨论外挂模块有效、隐蔽地与用户交互的方法。

在 Windows 平台下，交互技术通常有下面 4 种。

- 消息钩子。
- Hook 游戏消息处理过程。
- GetKeyState()、GetAsyncKeyState() 和 GetKeyBoardState() 函数。
- 进程间通信。

下面详细介绍这 4 种技术在外挂中的运用。

4.1　消息钩子

Windows 的消息机制允许用户插入消息钩子，以便消息在被消息处理过程处理之前被用户优先处理。消息钩子的设立和取消，由 Windows 提供的 SetWindowsHookEx() 和 UnhookWindowsHookEx() 这两个函数负责实现。

在用户态，Windows 的消息处理机制主要由以下 3 部分组成。

- 窗口处理过程。
- 线程消息队列。
- 线程消息循环。

我们知道，在 Windows GUI 编程中，要想建立一个窗口与用户进行交互，必须先定义一个窗口类 WNDCLASS。在这个结构体中，一个成员的初始化就是首先赋值窗口处理过程的地址，然后调用 CreateWindow() 函数创建一个或多个窗口，最后形成一个取消息（GetMessage() 函数）、转译消息（TranslateMessage() 函数）和分发消息（DispatchMessage() 函数）的消息循环。

因为 Windows 的整体消息机制比较复杂，不仅涉及用户态，还涉及对内核态的 Win32k.sys 模块的支持，所以，这里为了简单明了地阐述用户态的消息处理流程，只在用户态把窗口处理过程、线程消息队列、线程消息循环以及消息的流向抽象到图 4–1 中。

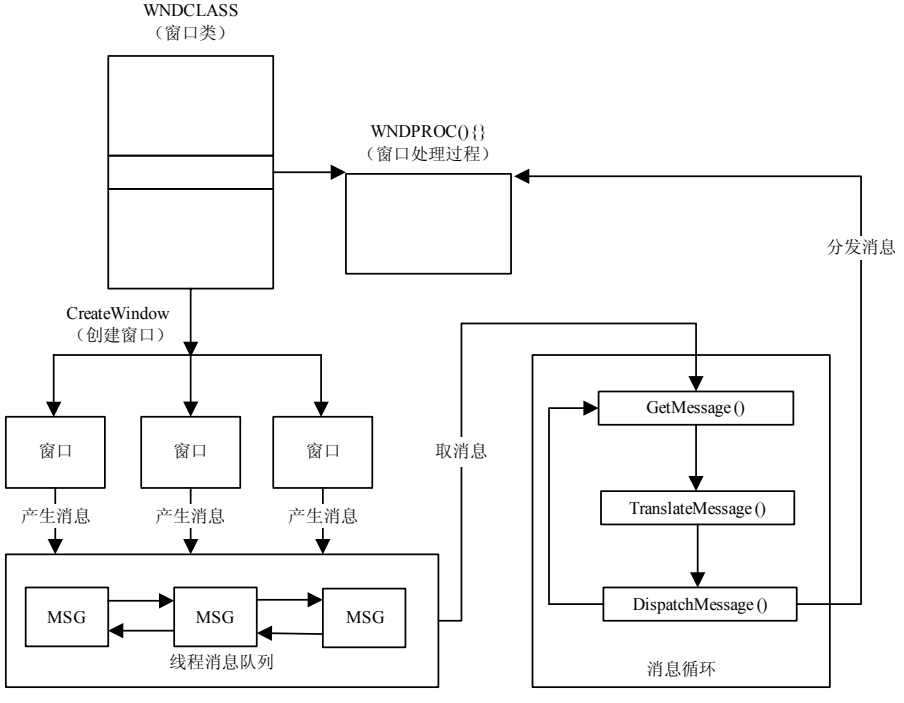


图 4-1 消息的逻辑分发过程

可以看到，窗口产生消息后，被系统分发至线程消息队列；线程在消息循环中通过 GetMessage() 函数从消息队列中读取消息，经过 TranslateMessage() 函数的转译，通过调用 DispatchMessage() 函数，最终把消息分发给 WNDPROC() 函数进行处理。而 SetWindowsHookEx() 函数插入的消息钩子，就是在将消息分发给 WNDPROC() 函数进行处理之前截获的。

下面，让我们看看当外挂模块注入游戏进程之后，如何通过 SetWindowsHookEx() 函数和 UnhookWindowsHookEx() 函数来实现设置和取消钩子。

```
    // 全局保存键盘钩子句柄，以备取消钩子时使用
    HHOOK g_hHook = NULL;
    // 拦截键盘消息处理过程
    BOOL    HookKeyboardMsgProc(DWORD dwThread)
    {
            BOOL  bSuccess = TRUE;
       HHOOK hhook;
       __try
       {
    // 注册消息钩子
    hhook = SetWindowsHookEx(WH_KEYBOARD,
     (HOOKPROC)HookKeyboardMsg,
    NULL,
    dwThread);
            if (hhook == NULL)
            {
    OutputDbgInfo(("[-] HookWndMsgProc Keyboard SetWindowsHookEx error
code %d!\n",
    GetLastError()));
               bSuccess = FALSE;
            }
         // 保存 Hook 的句柄
         g_hHook == hhook;
       }
       __except(EXCEPTION_EXECUTE_HANDLER)
       {
```

```
        OutputDbgInfo(("[!] HookWndMsgProc Exception Exit...\n"));
        return FALSE;
    }
    return bSuccess;
}
```

上面的函数设立了一个键盘钩子，其对应的处理函数为 HookKeyboardMsg()，所有游戏窗口内的按键消息都会先步入该函数。下面就让我们看看这个函数的实现。

```
LRESULT CALLBACK HookKeyboardMsg(
                            int code,
                            WPARAM wParam,
                            LPARAM lParam
                            )
{
    LRESULT lResult = 1;
    __try
    {
    // 判断是否是外挂需要的按键消息
    if( wParam == 外挂需要处理的消息)
    {
    // 调用外挂所对应的功能函数，如“无敌”、“加速”等，以实现交互
    // 省略代码
// 直接返回，表示该消息已被处理，无须下发
return lResult;
}
// 把外挂不关心的消息下发，使 Hook 链上的其他钩子可以处理消息
// 只有这样，才能使游戏的窗口处理过程获取该消息
lResult = CallNextHookEx(g_hHook, code, wParam, lParam);

    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        OutputDbgInfo(("[!] HookKeyboardMsg Exception Exit...\n"));
        return -1;
```

```
    }
 return lResult;
}
```

4.2　替代游戏消息处理过程

当外挂模块注入游戏进程之后，可以通过 GetWindowLong() 和 SetWindowLong() 这两个 API 来获取和重置窗口处理过程。这两个函数的用法比较简单，就不详细介绍了，示例如下。

```
BOOL HookWindowMsgProc( LONG lNewWindowMsgProc)
{
  WNDPROC pGameWndProcAddr = NULL;
    BOOL bSuccess = FALSE;
    __try
    {
      // 获取游戏窗口句柄
      HWND  hCurrentWnd = GetForegroundWindow();
if (NULL != hCurrentWnd)
        {
// 获取游戏主窗口处理过程
pGameWndProcAddr = (WNDPROC)GetWindowLongA(hCurrentWnd, GWL_WNDPROC);
// 设置新的窗口处理过程
 SetWindowLong(hCurrentWnd, GWL_WNDPROC,  lNewWindowMsgProc);
bSuccess     =  TRUE;
        }

    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        OutputDbgInfo(("[!]HookWindowMsgProc Exception Exit...\n"));
    }
    return bSuccess;
}
```

4.3　GetKeyState、GetAsyncKeyState 和 GetKeyBoard State

GetKeyState()、GetAsyncKeyState() 和 GetKeyBoardState() 这 3 个函数都是与获取键盘按键状态相关的 Windows API，因为其用法简单，所以常被外挂所采用。

在详细讨论这 3 个函数的用法之前，让我们简单地了解一下 Windows 下的按键处理流程，如图 4–2 所示。

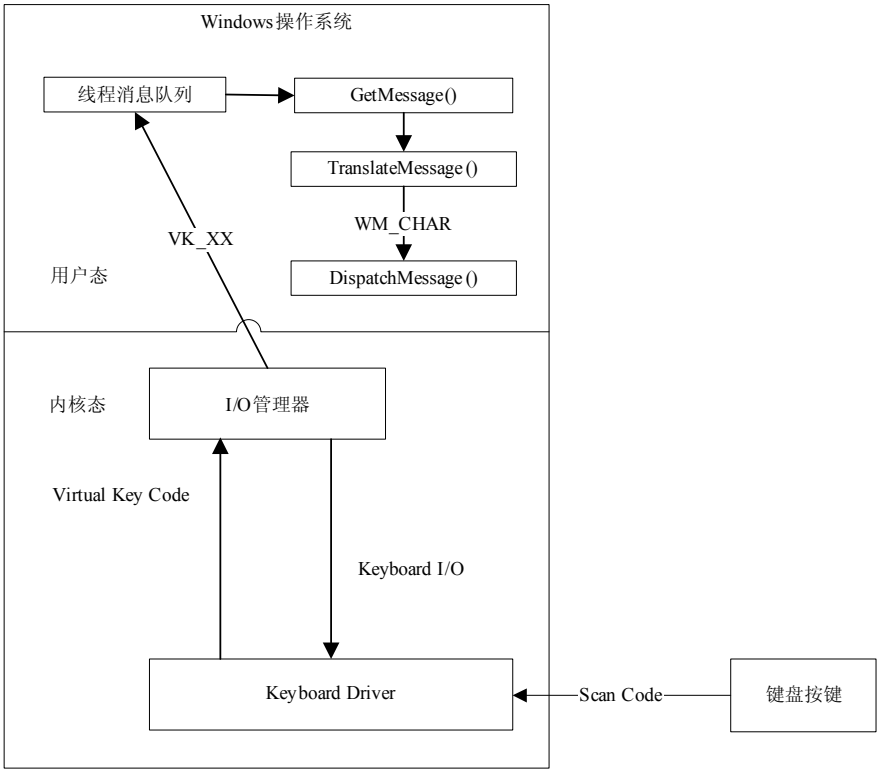


图 4-2　Windows 按键处理流程

当我们在键盘上按下任意一个键后，都会产生一个键盘中断。Keyboard Driver（键盘驱动）捕获这个中断后，会负责读取一个对应的 Scan Code（扫描码），同时将该 Scan Code 转化为 Virtual Key Code（虚拟键码），然后把这个 Virtual Key Code 放

入 I/O 管理器发出的空 IRP（中断请求包）中，并结束这个 IRP，返回用户态的 Windows 操作系统。在用户态，Windows 操作系统把这个 Virtual Key Code（即 VK_XX）包装成 MSG 的形式发送到对应的线程消息队列中，线程的消息主循环可以通过 TranslateMessage() 函数把 VK_XX 消息转换成 WM_CHAR 消息。

在上面的 Windows 键盘按键处理过程中有两个转换：一个是 Keyboard Driver 把 Scan Code 转换为 Virtual Key Code；另一个是 Windows 为了方便用户编程，把 Virtual Key Code 转换为对应的 WM_XX 消息，并把 WM_XX 消息和 Virtual Key Code 封装到 MSG 结构体中。例如，按下键盘上的【Tab】键，收到的消息 MSG.message 就是 WM_KEYDOWN，而 MSG.wParam 就是 VK_TAB。

有了对 Windows 键盘按键处理流程的了解，下面就让我们分析一下这 3 个函数在使用上有什么异同。

3 个函数的原型声明如下。

- SHORT：GetKeyState(int nVirtKey);
- BOOL：GetKeyBoardState(PBYTE lpKeyState);
- SHORT：GetAsyncKeyState(int nVirtKey);

3 个函数的相同点如下。

- 参数都是 Virtual Key Code 相关的。除了 GetKeyBoardState() 函数的参数是一个待系统返回的包含 256 个 Virtual Key Code 的缓冲器，其他两个函数都用于输入待查的 Virtual Key Code。
- 功能都是查询按键的状态。

3 个函数的不同点如下。

- GetKeyState() 函数只能在键盘消息处理过程中使用，如键盘消息钩子等，因为只有在线程读取消息队列中的键盘消息的时候，被查询的按键的状态才会发生改变。如果需要在消息处理过程以外查询按键的状态，就要调用 GetAsyncKeyState() 函数，通过返回值大于 0 或小于 0 来判断对应的按键是否按下。
- GetKeyBoardState() 函数与 GetKeyState() 函数的使用环境类似，也是只有当线程从消息队列中把键盘消息移除的时候，被查询的按键的状态才会发生改变。不过，在键盘消息处理过程中，调用该函数可以获取所有虚拟键的状态。

- GetAsyncKeyState() 函数是通过向底层键盘驱动发送查询请求来获取按键状态的，所以不存在必须在消息处理过程中使用的问题。其返回值表示两个内容：一是最高位是否为 1，代表是否按下该键；二是最低位是否为 1，表示上次调用该函数之后，这个按键是否被按下。

事实上，使用 GetAsyncKeyState() 函数能够实时获取按键的状态，局限性也比较小，其用法一般如下。

```
if( GetAsyncKeyState(VK_TA B) & 0x8000 )
{
   // 【Tab】键按下
}
```

但是，这 3 种与用户交互的方式比较容易被检测到——只要 Hook 对应的函数，就能很快定位外挂核心代码的地址。

4.4 进程间通信

Windows 操作系统提供了很多进程间通信的方法，如内存映射文件、管道、消息等。由于这方面的技术已经被研究得很透了，所以这里不再赘述。

本节将带领读者设计一个基于命名管道进行安全交互的程序。先看一下这个程序的逻辑结构，如图 4-3 所示。

假设外挂模块是一个 DLL，其中关于命名管道的代码分成两部分：一部分是实现管道 server，另一部分是实现管道 client。用于实现管道 client 的模块叫做外挂 client 模块，反之叫做外挂 service 模块。把外挂 service 模块和外挂 client 模块分别注入 Game 进程和 Service 进程，外挂 Service 进程将创建命名管道，等待管道 client 的连接。同时，当外挂 client 模块注入游戏进程之后，会打开命名管道，建立与管道 server 的连接。这样，一个在外挂 server 模块和外挂 client 模块之间的管道就架设好了。接下来，外挂 server 模块可以调用 GetAsyncKeyState() 函数或者通过其他方式获取用户输入的按键信息，然后通过命名管道将其传递给外挂 client 模块。

下面，我们从代码的角度来看看管道是如何创建、建立连接和传递数据的，以及如何构建管道 server 模块和管道 client 模块。

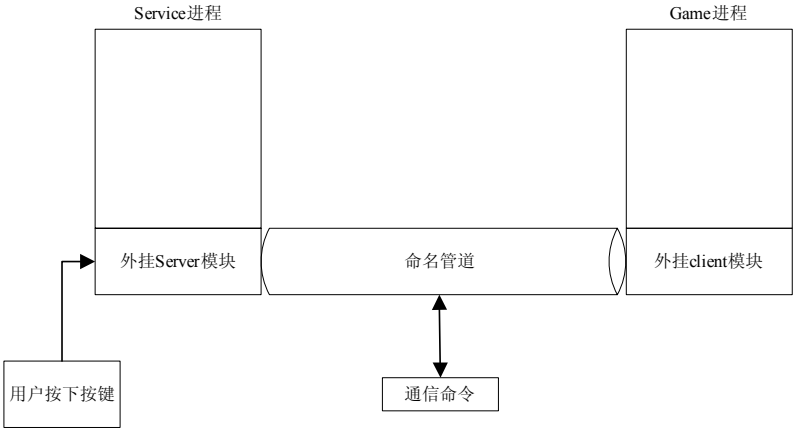


图 4-3　命名管道通信

（1）创建管道，示例如下。

```
HANDLE  OnPipeCreate(LPCTSTR pPipeName)
{
   HANDLE hPipe = NULL, hEvent = NULL;
   OVERLAPPED ovlap;
   hPipe = CreateNamedPipe(pPipeName,
PIPE_ACCESS_DUPLEX ,
    0,1,1024,1024,0,NULL);
   if(INVALID_HANDLE_VALUE == hPipe)
    {
      PRT(("[-] 创建命名管道失败！"));
      hPipe=NULL;
      return hPipe;
    }
    ConnectNamedPipe(hPipe,NULL);
}
```

（2）建立连接，示例如下。

```
bool OnPipeConnect(LPCTSTR pPipeName)
{
    bool   bRet = false;
```

```
        HANDLE hPipe = NULL;
        if(!WaitNamedPipe(pPipeName,NMPWAIT_WAIT_FOREVER))
        {
            PRT(("[-] 当前没有可利用的命名管道实例！"));
        }
        hPipe=CreateFile(pPipeName,GENERIC_READ | GENERIC_WRITE,
           0,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
if(INVALID_HANDLE_VALUE == hPipe)
        {
           PRT(("[-] NamedPipeOp:打开命名管道失败！"));
           goto Exit;
        }
        bRet = true;
    Exit:
        return bRet;
    }
```

（3）通过管道读取数据，示例如下。

```
    bool OnPipeRead(HANDLE hPipe,LPSTR pOutBuffer, ULONG uOutLen)
    {
        bool bRet = false;
        DWORD dwRead = 0;
        if(!ReadFile(hPipe,pOutBuffer,uOutLen,&dwRead,NULL))
        {
           PRT(("[-]读取数据失败！"));
           goto Exit;
        }
        bRet = true;
    Exit:
        return bRet;
    }
```

（4）将数据写入管道，示例如下。

```
    bool OnPipeWrite(HANDLE hPipe,LPSTR pInputBuf, ULONG uInputLen)
```

```
{
    bool bRet = false;
     DWORD dwWirte = 0;
     if(!WriteFile(hPipe,pInputBuf,uInputLen,&dwWirte,NULL))
   {
       PRT(("[-] NamedPipeOp: OnPipeWrite-WriteFile,写入数据失败! "));
       goto Exit;
   }
    bRet = true;
Exit:
    return bRet;
}
```

（5）创建管道 server 模块，示例如下。

```
// 创建管道
hPipe  = OnPipeCreate("\\\\.\\pipe\\WinsunPipe");
if(!hPipe)
    {
        PRT(("[-] Create Named pipe is error!\n"));
        return 0;
    }

// 等待 clinet 端关闭管道
hNamedEvent = CreateEvent(
NULL,
                    false,
                    false,
                    "PipeDataEvent"
                        );
if(!hNamedEvent)
    {
      PRT(("[-] Create Named event is error!\n"));
      return 0;
 }
```

```
// 循环接收用户输入，并把数据传递给管道 client 模块
while(TRUE)
{
   // 判断管道 client 模块是否关闭
   if(WAIT_OBJECT_0 == WaitForSingleObject(hNamedEvent, 0))
{
// 管道 client 模块已经关闭
// 关闭管道 server 端
DisconnectNamedPipe(hPipe);
// 省略其他代码
}
else
{
      // 判断用户是否按下按键，请求外挂提供相应功能
      if( GetAsyncKeyState if( GetAsyncKeyState(VK_TA B) & 0x8000 )
{
        // 【Tab】键按下
        // 数据写入管道
         OnPipeWrite(hPipe,待写入数据地址,待写入数据长度);
        FlushFileBuffers(hPipe);
}
}
}
```

（6）创建管道 client 模块，示例如下。

```
// 判断管道是否可以连接
if( hPipe = OnPipeConnect("\\\\.\\pipe\\WinsunPipe") )
    {
      // 连接管道 server 模块以后，读取管道数据
     while(1)
     {
       OnPipeRead(hPipe, 接收数据 buffer, buffer 长度);
     }
    }
```

由上面的代码我们可以知道，通过命名管道实现交互的方法比较简单，思路也比较清晰。

4.5　本章小结

本章主要讨论如何将用户的输入传递给外挂模块。虽然从形式上看，本章提供的是 4 种技术，但实际上，其原理还是集中在 Windows 提供的消息机制和进程间通信机制方面。希望本章讨论的这 4 种交互方式，能够帮助读者了解外挂在交互上可能采用的技术，以及各种交互技术的特点。

第 5 章　未授权的 Call

未授权的 Call 即通常所说的 Call 函数，它一向是外挂的核心功能。如何定位游戏关键函数，以及如何防止 Call 函数的行为被检测到，都是编写一款外挂时需要重点研究和关注的。与此同时，作为一名安全工程师，也只有深刻理解 Call 函数所涉及的技术，才能更好地检测和防御 Call 函数的这种“作弊”行为。所以，本章将把讨论的重点放在如何防止 Call 函数被检测到以及如何定位游戏关键函数上。

5.1　Call Stack 检测

Call Stack 就是某个函数被调用时的栈帧信息。Call Stack 检测就是利用栈帧信息中的调用返回地址来确定发起调用的地址是否属于合法的地址区间。更多有关 Call Stack 检测的主题，参见第 10.2 节。

5.2　隐藏 Call

事实上，单纯从技术的角度看，Call 一个游戏函数来实现某项特定的外挂功能是很简单的——构造函数执行环境（如正确的 ECX 值），传入必要的参数，直接 Call 就是了，其难点在于 Call 游戏函数之后不被检测到。

下面就让我们了解一下能够更好地隐藏 Call 函数行为的两种方法。

5.2.1　Call 自定义函数头

假设我们要 Call 游戏中的 MessageBox() 函数。MessageBox() 函数的原型如下。

```
int MessageBox(
    HWND hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT UType
    );
```

在游戏进程中，获取 MessageBox() 函数地址的方法很多，比较常用的方法是通过 GetModuleHandle() 函数和 GetProcAddress() 函数获取。假设现在 MessageBoxA() 函数在内存中的地址是 ADDR_MSGBOX，Call MessageBox() 函数的汇编伪代码通常如下。

```
  push UType
  push lpCaption
  push lpText
  push hWnd
  call  ADDR_MSGBOX
```

虽然上面这个 Call 函数的实现方法很简单，只要向栈中压入对应数目的参数，然后 Call 目标函数地址就可以了，但是“call ADDR_MSGBOX”语句很容易被检测和分析——因为只要在 ADDR_MSGBOX 代码的开始处设立一个钩子，进行 Call Stack 检测，就能快速发现 Call 函数的行为。

我们不妨做一个实验，道具如下。

- Notepad.exe 进程（暂时代替游戏进程 Game.exe）。
- 分析工具 GameSpider.dll（第 7.1.1 节会专门介绍）。
- 分析游戏的开源工具 Cheat Engine.exe。

实验步骤如下。

（1）打开 Notepad.exe，将分析工具 GameSpider.dll 注入 Notepad.exe。如图 5-1 所示，Notepad.exe 模块的地址范围是 [B20000, B50000]。

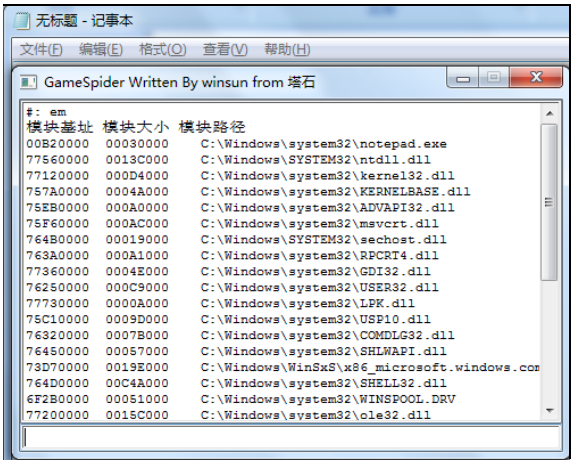


图 5-1　将 GameSpider 注入 Notepad.exe

（2）Hook MessageBox() 函数，设置 Call Stack 检测。当然，进行 Hook 之前要获取 MessageBoxA 的地址，在这里可以通过 ga 命令和导出函数的名字“MessageBoxA”来获取，如图 5–2 所示。通过 hac 命令对 0x762aea11（MessageBoxA 地址）设置 Detour Hook，如图 5–3 所示。

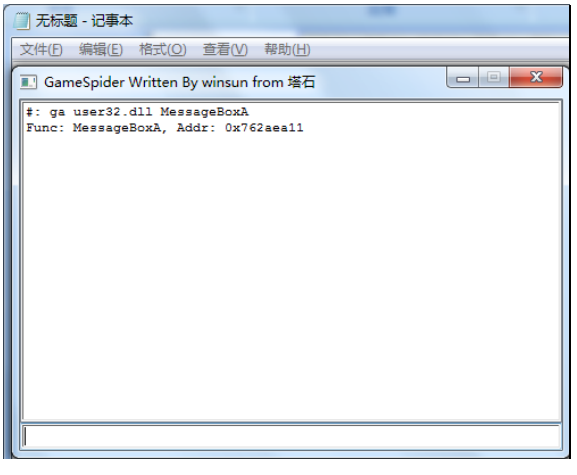


图 5-2　获取 MessageBoxA 的地址

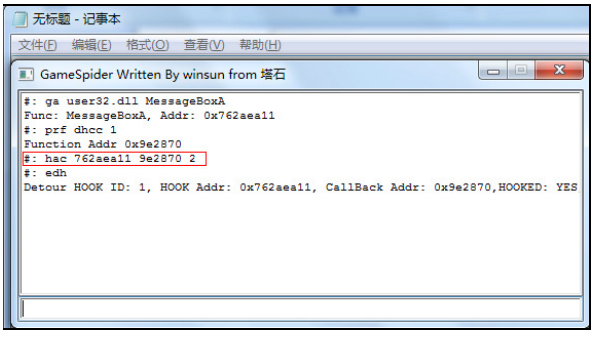


图 5-3　用 hac 命令设置 Detour Hook

（3）通过 Cheat Engine 的脚本引擎编写 Call MessageBoxA 函数功能代码，如图 5–4 所示。

（4）通过 Cheat Engine 脚本引擎远线程注入代码，执行 MessageBoxA 函数，如图 5–5 所示。可以看到，CallMessageBoxA 这个标签的地址是 0x00A10000。

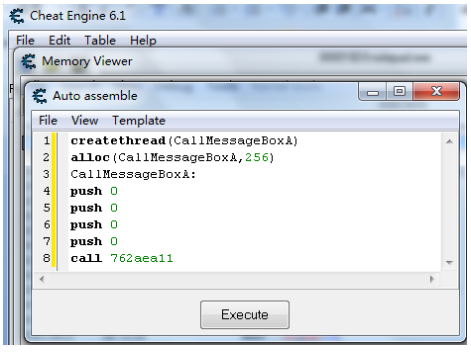


图 5-4　用 Cheat Engine 编写 Call MessageBoxA 代码

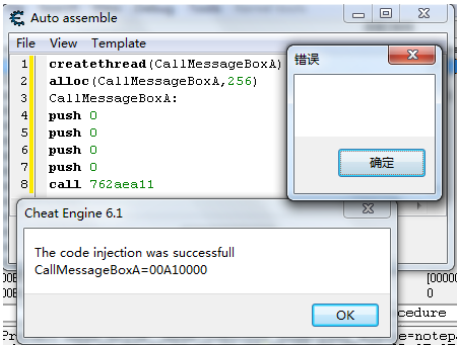


图 5-5　Call MessageBoxA 执行后弹出的对话框

（5）Call MessageBoxA 的行为被检测出来，并输出结果到 DebugView，如图 5–6 所示。

从图 5–6 中可以看到，调用 MessageBoxA 后返回的地址是 0xa1000d。这个返回地址既不在 Notepad.exe 模块的地址范围 [B20000, B50000] 内，也不在 Notepad.exe 进程空间其他模块的地址区间内，所以，MessageBoxA 并不是被 Notepad.exe 模块所 Call，而是被隐藏模块或注入的代码所 Call。由图 5–4 可知，返回地址位于标签

CallMessageBoxA 所在地址之后 13 字节的地方，这个地方就在“Call 762aea11”指令之后，如图 5-7 所示。

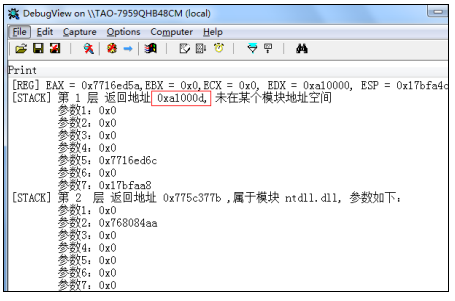


图 5-6　MessageBoxA 的 Call Stack 检测结果

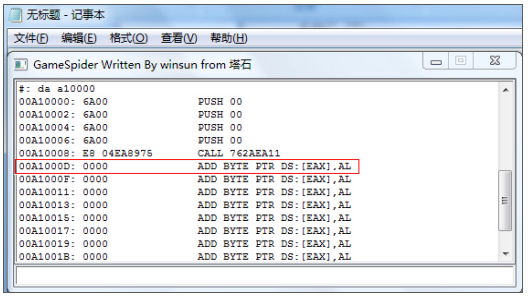


图 5-7　返回地址 0x00A100D

综合来看，0xA10000 开始的代码是 Cheat Engine 通过远线程注入 Notepad.exe 进程的代码，这段代码的作用是调用 MessageBoxA，但是它却被我们设置在 MessageBoxA 函数处的 Call Stack 检测检测到了，如图 5-6 所示。同样的道理，如果 Cheat Engine 的这个简单的 Call MessageBoxA 过程放在一个带有 Call Stack 检测的程序中，同样很快会被发现。

外挂 Call MessageBoxA 函数的内存示意图如图 5-8 所示。

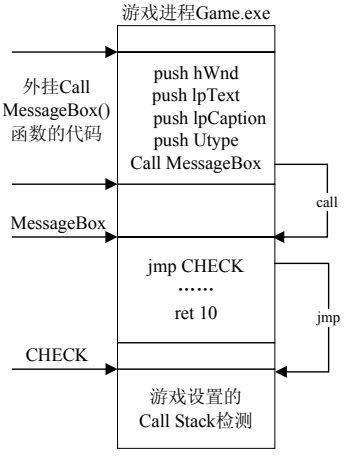


图 5-8　外挂 Call MessageBox 函数

MessageBox() 函数的开头有一个 jmp 指令，这个指令跳转到了游戏设置的针对该

函数进行的 Call Stack 检测代码中。所以，当外挂直接 Call MessageBox() 函数的时候，就会被 Call Stack 检测代码检测到。有什么办法可以绕过这里的 Call Stack 检测呢？这就是本节要解决的问题—— Call 自定义函数头。

如图 5-8 所示，只要 EIP（程序执行指针）从 MessageBox() 函数的头部开始往下执行，那么 EIP 就一定会跳转到游戏设置的 Call Stack 检测代码中。如果不想跳入 Call Stack 检测代码，只要不执行“jmp CHECK”指令即可。如果既要 Call MessageBox() 函数，又要确保不执行“jmp CHECK”指令，该怎么做呢？答案是自定义函数头—— Call 自定义函数头。

自定义函数头就是从原始函数头开始复制若干个字节（或若干条指令）到另一个地址处，构成新的函数头，复制的条件是复本长度要大于“jmp CHECK”指令的长度。需要在新函数头的末尾添加跳转指令，以保持原始函数的指令执行流程。之后再 Call 新函数头时，就会绕过原始函数头设下的 Hook，进而避开 Call Stack 检测代码的检测，同时顺利执行原始函数的功能。

Call 自定义 MessageBoxA 函数头的内存示意图如图 5-9 所示。

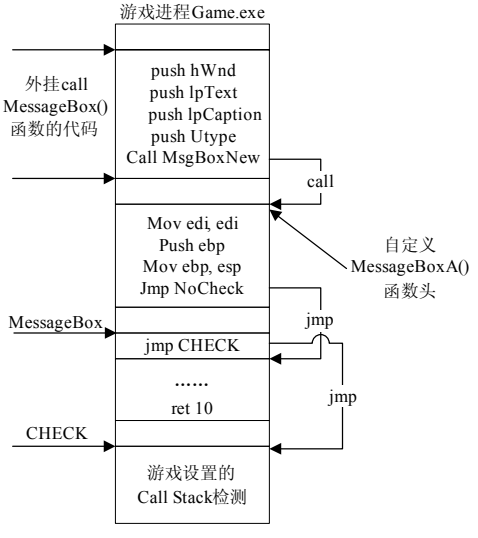


图 5-9　Call 自定义 MessageBoxA 函数头

从图 5-9 中可知，当 Call MsgBoxNew 被调用之后，EIP 的执行流程是：先执行 MessageBoxA 函数的自定义函数头，再跳入“jmp CHECK”指令之后的指令开始执行。

所以，整个流程既躲避了 Call Stack 检测，又达到了 Call MessageBoxA 函数的目的。

事实上，在真正的对抗中，情况可能比这里描述的还要激烈。在真实的环境下，反外挂所加的 Hook，可能不在某个函数的开头——可能在末尾，也可能在中间。所以，外挂作者需要具体情况具体分析。但是，绕过的原理基本都是一致的。不过，虽然好不容易逃过了函数头的 Call Stack 检测，却还是难逃多层堆栈检测的厄运——真是“逃得了和尚，逃不了庙”。什么是多层堆栈检测呢？大家可以先思考一下。

我们调用一个函数以取得某项功能的时候，在这个函数内部还有可能继续调用其他函数——在函数内调用函数，一直嵌套调用下去，形成一个调用链。所以，在某个运行时刻，任意一个得到调用的函数一定处在某个固定的调用链上，我们可以向上或向下跟踪它的调用情况。调用 MessageBox() 函数的伪代码示例如下。

```
int main(void)
{
    int iRetValue = -1;
    iRetValue = CallOutputMessage();
    return iRetValue;
}
int CallOutputMessage()
{
   int iCallSuccess = 1;
   MessageBox(NULL, "winsun", "winsun", MB_OK);
   return iCallSuccess;
}
```

上面这个 main() 函数调用了 CallOutputMessage() 函数，而 CallOutPutMessage() 函数又调用了 MessageBox() 函数，从而使 main()、CallOutputMessage()和 MessageBox() 这 3 个函数构成了一个调用链。

这个简单的调用链如图 5-10 所示。与图 5-10 对应的栈帧情况如图 5-11 所示（进入 MessageBox() 函数后的栈帧）。

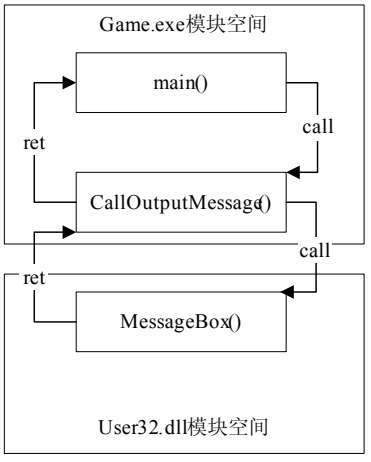


图 5-10　函数调用链

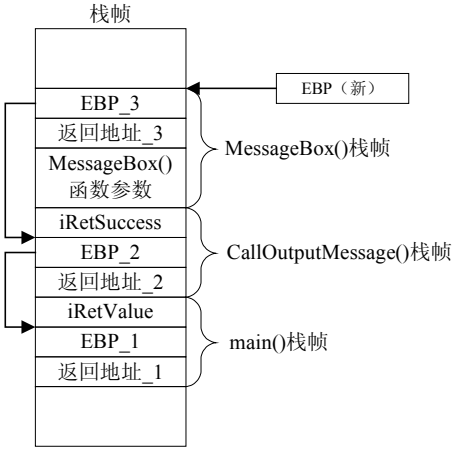


图 5-11　进入 MessageBox() 函数后的栈帧

多层堆栈检测方法就是 Hook 调用链低层（如图 5-11 中的 MessageBox() 函数处）的函数或地址，然后进行多层堆栈回溯检测。堆栈回溯检测的判定规则是判断返回地址是否属于合法模块，一旦发现返回地址不属于合法模块，立即判定其为非法。根据如图 5-11 所示的栈帧情况，假设我们已经 Hook 了 MessageBox() 函数，下面的伪代码展示了如何进行堆栈回溯检测。

```
    void StackFrameBackCheck( int iCheckLevel ) // iCheckLevel 检测栈帧
的层数
    {
    DWORD dwCurrentEBP = 0;             // 当前 EBP
    DWORD dwRetAddr = 0;                // 函数返回地址
    PDWORD pdwCurrentEBP = NULL;        // 指向存放当前 EBP 的指针
       __asm{
          mov [dwCurrentEBP], ebp       // 获取当前 EBP
    }
    pdwCurrentEBP = & dwCurrentEBP;
    for( int iLoop = 0; iLoop < iCheckLevel; iLoop++ )
    {
        // 逆调用方向，判定返回地址的合法性
        // 获取当前返回地址
```

```
dwRetAddr = *(pdwCurrentEBP + 1);
// 对返回地址进行合法性判定
{
   if( dwRetAddr < 游戏模块的基地址
|| dwRetAddr > (游戏模块的基地址+游戏模块大小) )
{
  // 判定为非法模块
  return;
}
}
// 获取上一个栈帧的 EBP 地址，准备上一层堆栈检测
pdwCurrentEBP =  (PDWORD) *pdwCurrentEBP;
}
}
```

多层堆栈检测的优点在于：就算外挂躲过了某一层的堆栈检测，但由于检测是多层的，所以只要是来自不正常模块的调用，始终会在某一层被检测到。而且，对反外挂系统来说，可以选择一个较低层进行多层堆栈检测，这样就一劳永逸了。

一个外挂 Call 自定义函数被多层堆栈检测的示意图如图 5-12 所示。

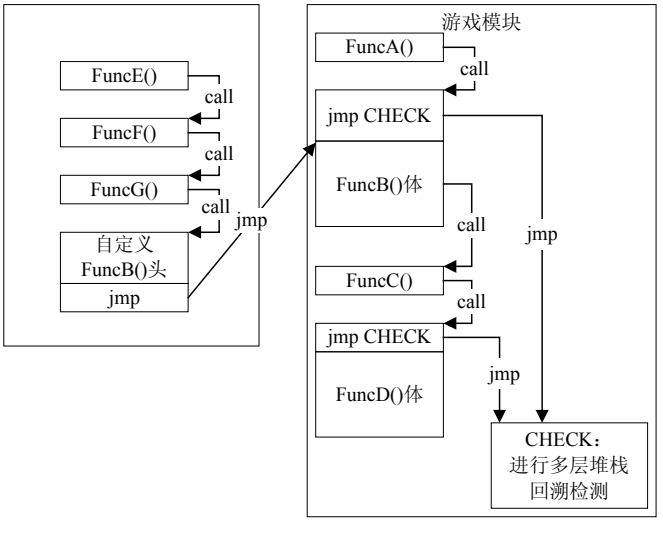


图 5-12　多层堆栈检测 Call 函数

在图 5-12 中，外挂模块中的 FuncG() 函数采用 Call 自定义函数头的方式 Call 游戏模块中的 FuncB() 函数。虽然这种 Call 自定义函数的方式躲过了游戏模块 FuncB() 函数中设置的多层堆栈回溯检测，但是，由于调用链的低层有 FuncD() 函数，反外挂系统仍然在此添加了堆栈回溯检测，所以，外挂的此次 Call 行为还是会被检测到。有些读者可能会说："我们再用 Call 自定义函数头的方式进行 FuncD() 函数的堆栈回溯不就可以了吗？"但是，在实际对抗中，可不是那么容易知道反外挂系统到底在低层的哪个地方设置了多层回溯检测。既然这样，我们是不是没有办法进行多层堆栈检测了呢？其实，躲避检测的办法还是有的，第 5.2.2 节讲的就是这个问题。

5.2.2　构建假栈帧

根据第 5.2.1 节对多层堆栈回溯检测原理的分析可以知道，检测规则是判定返回地址的合法性，而返回地址来源于 [EBP+4]。如果在 Call 目标函数之前给 EBP 赋一个假的栈基地址，那么可想而知，通过 [EBP+4] 获取的返回地址肯定是假的。

下面我们一起设计一个通过构建假栈帧来躲过多层堆栈回溯检测的方案。

CallTest.exe 程序的核心代码如下。在这个程序中，经过 4 层函数的调用，最终实现了 Call MessageBoxA() 函数。在这里，需要事先在 MessageBoxA() 函数中设置一个 4 层堆栈回溯检测的钩子。

```
void FuncE(void);
void FuncF(void);
void FuncG(void);
void FuncE(void)
{
    FuncF();
}

void FuncF(void)
{
    FuncG();
}
void FuncG(void)
```

```
    {
       MessageBox(NULL, "Test Bypass Call Stack Check", "winsun", 0);
    }
    WinMain( HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nShowCmd )
    {

        MessageBox(NULL, "Select Ok For Test Bypass Call", "winsun",
MB_OKCANCEL);
        FuncE();
    }
```

CallTest.exe 程序是 Visual C++ 6.0 编译环境下的一个 Windows Application，它是 GUI 程序，不是 Console 程序。

运行 CallTest.exe 程序的效果如图 5-13 所示，这个界面为我们 Hook MessageBoxA() 函数提供了一个时机。打开这个界面时候，我们可以先注入分析工具，如 GameSpider，然后对 CallTest.exe 进程中的 MessageBoxA() 函数设立一个 4 层堆栈检测。

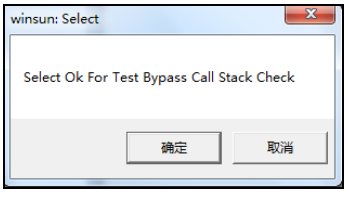


图 5-13　选择是否测试 Bypass Call Stack Check

Hook MessageBoxA() 函数，并对此函数设置一个 4 层堆栈回溯检测，如图 5-14 所示。这个时候，CallTest.exe 模块的地址范围是 [400000, 425FFF]，如图 5-15 所示。

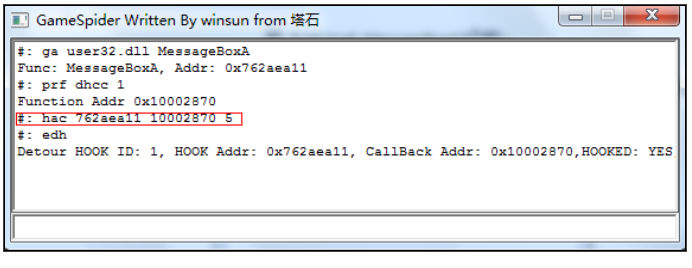


图 5-14　Hook MessageBoxA() 函数

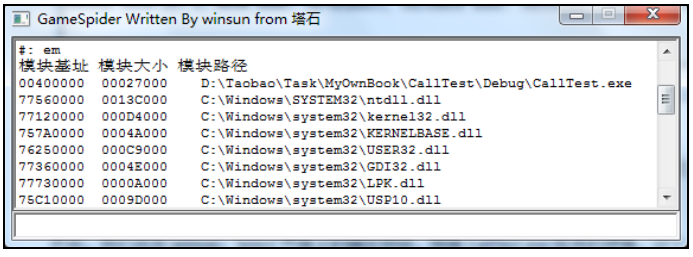


图 5-15　模块地址范围

单击“winsun: select”界面上的“确定”按钮，根据 CallTest.exe 程序的逻辑，这时会调用 MessageBoxA() 函数。此时，会弹出另一个对话框“winsun: Bypass”，如图 5-16 所示。

由于之前我们设置了 Hook，这时 DebugView 的界面中会显示一个 6 层堆栈回溯检测结果，如图 5-17 所示。

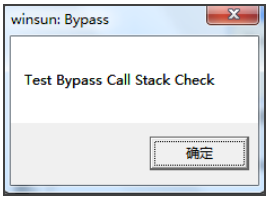


图 5-16　winsun: Bypass 消息框

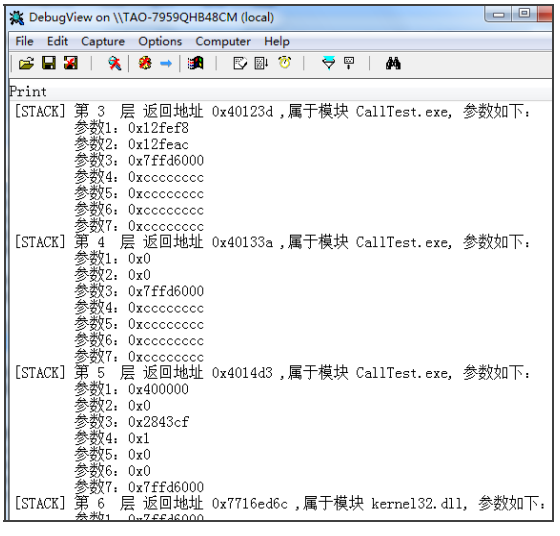


图 5-17　6 层堆栈回溯信息

从图 5-17 的堆栈回溯信息中可以发现，前 5 层栈帧的返回地址都在 CallTest.exe 模块的地址范围中，而第 6 层的返回地址在 kernel32.dll 模块的地址空间中。为什么经过 6 层才回溯到 kernel32.dll？因为调用 WinMain() 函数的函数在第 5 层，且属于 CallTest.exe 进程。

接下来我们设计一个方案，使这里显示的栈帧的返回地址是任意值。

在设计这个方案之前，让我们先从逻辑上看看 CallTest.exe 的调用和栈帧示意图，如图 5–18 所示。

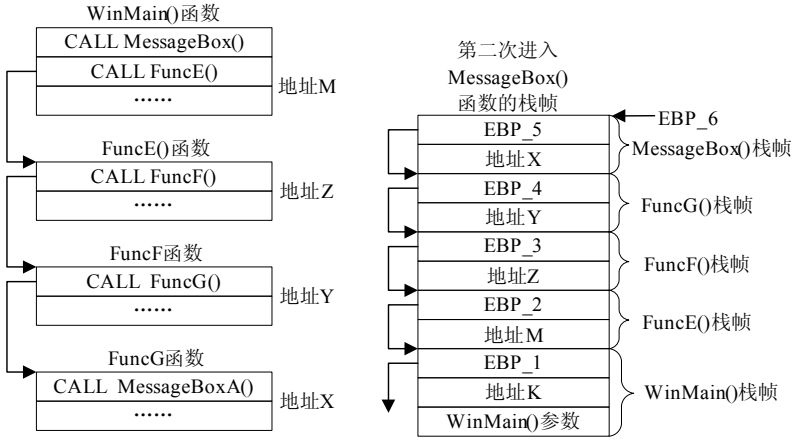


图 5-18　CallTest.exe 的调用和栈帧示意图

图 5–17 中的栈帧返回地址与图 5–18 右边部分中的返回地址一一对应：第 1 层返回地址对应地址 X，第 2 层返回地址对应地址 Y，第 3 层返回地址对应地址 Z，第 4 层返回地址对应地址 M，第 5 层返回地址对应地址 K。

图 5–18 的右边部分是 EIP 进入 MessageBoxA() 函数后的栈帧图。在这幅栈帧图中，我们可以发现如下推导。

- 地址 X=[EBP_6 + 4]
- 地址 Y=[EBP_5 + 4]
- 地址 Z=[[EBP_4] + 4]
- 地址 M=[[[EBP_3]] + 4]
- 地址 K=[[[[EBP_2]]] + 4]

根据推导出来的返回地址的公式，要想让这些返回地址成为反外挂系统认为合法的地址，只需要做以下两件事。

- 在跳入 MessageBox() 函数之前，给栈地址 [EBP_6 + 4] 赋予一个假的地址。
- 在跳入 MessageBox() 函数之前，给 EBP_5 赋予一个假的栈基地址。

因为 CallTest.exe 程序是在 FuncG() 函数中调用 MessageBox() 函数的，所以，在 FuncG() 函数的开头，即跳入 MessageBox() 函数之前的部分，加上一些处理动作，就可以使程序在跳入 MessageBox() 函数之前满足上面的条件。

现在，让我们看看新的 FuncG() 函数的实现内容，示例如下。

```
    void FuncG(void)
    {
    BypassCallStackCheck();    // 负责构建假栈帧
         MessageBox(NULL, "Test Bypass Call Stack Check", "winsun:
Bypass", 0);
    }

    void BypassCallStackCheck(void)
    {
       DWORD      dwFakeStackFrame[260] = {0};     // 假栈帧数组
       DWORD   dwFakeRetAddr = 0xFEFEFEFE;         // 假返回地址
       PCHAR    pszTitle = "ByPass Call Stack Check \0";
       PCHAR    pszCaption = "winsun \0";
       DWORD   dwMsgBoxAddr = 0;       // MessageBoxA()函数地址
       DWORD   dwXAddr = 0x7c921224;   // 地址 X，一个合法的地址，该地址处的
指令为 ret

    // 获取 MessageBoxA()函数的地址
    dwMsgBoxAddr = (DWORD)GetProcAddress(GetModuleHandle("user32.dll"),
"MessageBoxA");

      // 初始化假栈帧数组，这里为了演示方便，暂时初始化 50(100/2)个栈帧
      for ( int iLoop = 0; iLoop < 100; )
       {
          // 模拟 push EBP 操作，保存之前 EBP 的值
          dwFakeStackFrame[iLoop]  = (DWORD)&dwFakeStackFrame[iLoop+2];
    // 置入虚假的返回地址
    dwFakeStackFrame[iLoop+1] = dwFakeRetAddr;
            iLoop += 2;
```

```
        }
        // 准备 Call MessageBox()函数
        Bypass_Call_Func(dwFakeStackFrame,   dwMsgBoxAddr,   pszTitle,
pszCaption, dwXAddr);
    }
    // Bypass_Call_Func()函数是一个裸函数，负责真正构建假栈帧
    _declspec  (naked)  void  Bypass_Call_Func(DWORD    (&dwFakeStack
FrameArray)[260] , DWORD dwMsgBoxAddr, PCHAR pszTitle, PCHAR pszCaption,
DWORD dwXAddr)
    {
        _asm
        {
            // 函数前导指令
            mov edi, edi
            push ebp
            mov ebp, esp

            // 函数体
            mov  dwFakeStackFrameArray[259], ebp    // 暂时保存真实的 EBP

            jmp PUSH_REAL_RET_ADDR
    CALL_MSGBOX:
            push 0
            push pszCaption
            push pszTitle
            push 0
            push dwXAddr    // 将合法地址处的某个 ret 指令所在的地址压入栈中
            mov  eax, dwMsgBoxAddr
            mov  ebp, dwFakeStackFrameArray  // 把假栈帧基地址赋值给 EBP
            jmp  eax         //跳入 MessageBox()函数执行
    PUSH_REAL_RET_ADDR:
            // 将下面 REAL_RET_ADDR 处的地址压入栈中
            // 也就是说，将调用 MessageBox()函数后的真正返回地址压入栈中
            call CALL_MSGBOX
```

```
    REAL_RET_ADDR:
            mov ebp, dwFakeStackFrameArray[259]  // 获取之前保存的真实 EBP

            // 函数后继指令
            mov esp, ebp
            pop ebp
            ret 20

        }
    }
```

我们通过上面的 BypassCallStackCheck() 函数和 Bypass_Call_Func() 函数构建了一个假栈帧链。在这种情况下，Call MessageBox() 函数返回的栈帧检测信息会是什么样子呢？从图 5-19 中可以看到，现在栈帧读取的返回地址都是之前构建的假地址，所以，塞入什么样的假地址，Call Stack 检测就读取什么样的地址，完全不会影响程序的正常执行。

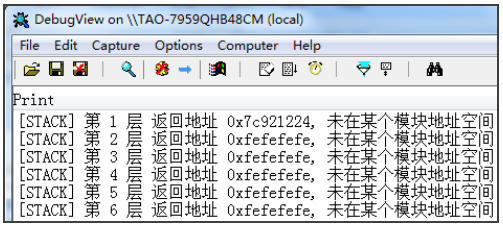


图 5-19　构建假栈帧后的 Call Stack 检测信息

BypassCallStackCheck() 函数和 Bypass_Call_Func() 函数的具体实现如图 5-20 所示：左边是正常的栈帧情况，右边是经过处理过的假栈帧链。仔细对比不难发现，变化的地方如下。

- 左边的“EBP_5”变成了右边假栈帧数组的首地址“&dwFakeFrame[0]”。
- 地址 X 变成了一个合法地址 0x7C921224（为了方便，采用硬编码），这个合法地址的指令是 ret。
- 在压入 MessageBox() 函数之前先压入真实的返回地址 REAL_RET_ADDR。

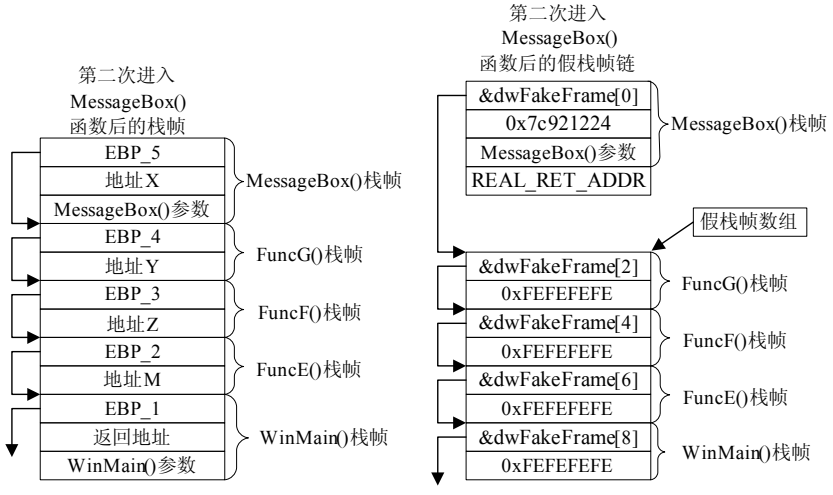


图 5-20　真假栈帧链对比

经过上面的变化之后，整个假栈帧构建完成。当进入 MessageBox() 函数执行之后，Call Stack 检测获取的第一个返回地址 0x7C921224 是一个合法地址。然后，根据检查算法，检测程序开始递归遍历假栈帧数组里由我们预先塞入的假地址。所以，到目前为止，我们成功地绕过了 Call Stack 检测。

虽然已经绕过了 Call Stack 检测，但是为了帮助读者更好地理解这种方法，下面分析一下执行 MessageBox() 函数之后，其返回的执行流程是什么样子的。

当 MessageBox() 函数执行到它的 Function Epilogue（函数后继指令）的时候，执行“mov esp，ebp”和“pop ebp”指令之后，ESP 将指向 0x7C921224 所在的栈地址；执行“ret 16”指令后，ESP 指向 REAL_RET_ADDR 所在的栈地址，EIP 被赋值为 0x7C921224；之后，当 EIP 从 0x7C921224 处取出 ret 指令执行时，EIP 被赋值为 REAL_RET_ADDR，程序的控制权再次回到 Bypass_Call_Func() 函数的 REAL_RET_ADDR 处。

这一套构建假栈帧的机制已经讲解完了。当然，其中还有很多细节可以采用其他方式实现，例如在合法地址空间寻找 ret 指令。其实，也可以通过在游戏代码段或数据段中插入指令序列“push/pop/ret”来代替。相信这套绕过 Call Stack 检测的方案已经比较完美了，读者可以在掌握之后，继续发挥想象力，构建更出色的方案。

5.3　定位 Call

在第 5.2 节中，我们已经详细讨论了如何巧妙地避开 Call Stack 检测。现在，让我们看看如何定位 Call。定位 Call，就是大家可能比较熟悉的找 Call。那么，到底什么游戏函数值得去找、去定位呢？当然是游戏中执行关键动作的一些函数，如施放技能的函数、召唤道具的函数、获取或设置坐标的函数、加红/减红的函数等。只要定位这些函数，我们就可以模拟函数执行环境，直接 Call 函数地址，以实现相应的功能。

定位 Call 函数是一项非常考验耐心和技术的工作，网络上流传的资料比较零散，也比较少。本节将通过虚函数差异调用和 send() 函数回溯这两种方法，系统地介绍如何降低定位 Call 的难度。

其实，在安全领域，差异思想是一种被广泛运用的经典思想，例如反 Rootkit 领域的检测进程隐藏功能（对比用户态枚举进程和内核态枚举结果）、Cheat Engine 中的搜索数据功能等。

下面就让我们从虚函数差异调用和 send() 函数回溯这两个方面来分别讨论如何定位 Call。

5.3.1　虚函数差异调用定位 Call

目前，很多 Windows 游戏客户端采用 C++ 语言开发。C++ 是一种面向对象语言，面向对象的封装、继承和多态特征蕴藏在游戏代码的设计中。在游戏中，多态是通过虚函数实现的。关于 C++ 语言的具体知识，这里就不阐述了，读者可以参考相关书籍。我们需要了解的是：在游戏中，许多功能函数是以虚函数的形式存在的，如设置/获取坐标函数、施放技能函数等。当然，还是要具体情况具体分析，例如，本节讨论的通过虚函数差异调用来定位 Call 的方法就适用于大量采用虚函数的游戏。

在游戏中，一个事件的发生，如捡起一个物品、加红、加蓝、碰撞精灵、砍怪等，都会引发一连串的函数调用（虚函数和非虚函数调用）。这里讨论的虚函数差异调用是指事件发生前后瞬间虚函数调用的差异。

例如，角色对象在捡起一个物品前，会调用一系列虚函数。假设这个系列虚函

数是集合 A(X_1, X_2, X_3, X_4, …, X_n)。角色对象捡起一个物品的瞬间，会调用一系列虚函数，假设这个系列虚函数是集合 B(X_1, X_2, X_3, Y_1, …, Y_n)。这里的 X 和 Y 都是虚函数地址。现在，角色捡物品之前和捡起瞬间，分别对应集合 A 和集合 B。通过对这两个集合的比较，可以发现 X_1、X_2 和 X_3 在两个集合中都存在，这样可以判断这 3 个虚函数对捡起物品这个动作意义不大，我们的侧重点就放在集合 B 中存在而在集合 A 中不存在的虚函数调用上。这种差异思想可以排除游戏中很多烦人的、非关注点的函数调用。

接下来，让我们看看设计虚函数差异调用的方案。首先，当然是监控游戏对象的虚函数调用。其次，要建立一个分析模型，以便在监控到虚函数调用后，筛选此虚函数是否需要重点关注。很多读者可能觉得监控游戏对象的所有虚函数调用是一项不太可能完成的任务，因为虚函数表的长度无法知道，因此，在第 6 章中，笔者将给出一套可行的完整 Hook 虚函数的方案。

下面我们来设计一个分析模型。在这个模型中，需要使用块内存来保存集合 A 和集合 B 的调用信息，还需要一个寻找差异的逻辑。在这个分析模型中，笔者采用比较节约内存的位图结构来映射虚表，以表示集合 A 和集合 B 的调用信息。为了说清楚位图的作用，先让我们看看游戏对象和虚表的内存关系，如图 5-21 所示（这里没有考虑由继承带来的对象嵌套和多个虚表）。

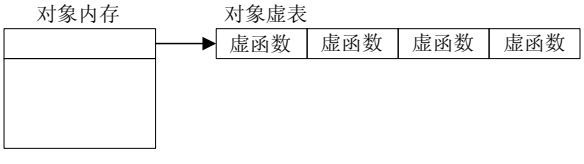


图 5-21　对象与虚表的关系

正是由于虚表的首地址存放在对象首地址的 4 字节中，所以虚函数的调用比较有规律。虚函数的调用形式一般如下。

```
mov eax, [ecx]      // ECX 是对象首地址，取出虚表首地址放入 eax
call [eax + 虚函数在虚表中的偏移]      // 调用虚函数
```

从上面的两条汇编指令可以看出，虚表其实就是一个未知大小的数组，数组成员的放置位置就是虚函数地址。

在第 6 章中，笔者采用的假虚表替换机制不仅能感知虚表中所有虚函数的调用行为，而且能感知被调用的虚函数来自虚表这个大数组中的哪一格，即感知数组下标。假设虚表是数组“DWORD dwVirtualTable[UNKNOW_SIZE]”，而位图是数组“DWORD dwBitGraph[VERY_BIG]”，就可以在这两个数组之间建立以下映射关系。

- dwVirtualTable[0] 对应于 dwBitGraph[0]：第 1 个 bit 位
- dwVirtualTable[1] 对应于 dwBitGraph[0]：第 2 个 bit 位
- dwVirtualTable[2] 对应于 dwBitGraph[0]：第 3 个 bit 位
- dwVirtualTable[3] 对应于 dwBitGraph[0]：第 4 个 bit 位
- ……
- dwVirtualTable[7] 对应于 dwBitGraph[0]：第 8 个 bit 位
- dwVirtualTable[8] 对应于 dwBitGraph[1]：第 1 个 bit 位

根据上面的映射关系可知，一旦发现 dwVirtualTable[x] 中的虚函数被调用，就置 dwBitGraph[x/8] 的第 x%8 个 bit 位为 1，否则为 0。

所以，根据上面的映射规则，在如图 5-22 所示的位图中，dwVirtualTable[1]、dwVirtualTable[10]、dwVirtualTable[19]、dwVirtualTable[28] 这 4 个虚函数得到了调用。

	0	1	2	3	4	5	6	7
0	0	1	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0
2	0	0	0	1	0	0	0	0
3	0	0	0	0	1	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

图 5-22　虚函数调用位图

根据图 5-22 中的映射规则，模型的分析过程如下。

（1）事件发生之前，监控虚函数的调用，并把调用映射到图 5-22。

（2）触发事件发生，监控虚函数的调用，计算出要映射到位图中的位置。

（3）判断这个位置上的值是否为 1。如果为 1，则返回；如果为 0，则打印这个虚函数，这个虚函数就是需要关注的与此次事件的发生相关的一个调用。

通过这里的虚函数差异调用分析，会得出一个虚函数调用集合，这个集合中的

函数是按时间先后顺序出现的。之后我们要做的事情就是逐个排除集合中的函数了，方法如下。

（1）看看函数调用附近是否有明文字符串。

（2）尝试构建调用环境 Call 该函数，看看是否能实现对应的游戏功能。

（3）过滤该函数的调用，看看是否能屏蔽该游戏功能。

虚函数差异调用定位 Call 就讲到这里，希望读者能结合第 6 章中的虚函数 Hook 方案一起看。虽然不一定能准确定位 Call，但是这种思想能排除很多不必要的干扰。事实上，这是一种接近准确的模糊定位技术，可以大大缩小定位范围。

5.3.2 send() 函数回溯定位 Call

在 Windows 网络编程中，send() 函数和 WSASend() 函数都是 ws2_32.dll 的导出函数，主要负责发送数据包。网络游戏客户端要与服务器通信，必须调用 send() 函数或 WSASend() 函数。因为基于发包函数来定位 Call 的思路是一致的，所以本节主要讨论基于 send() 函数回溯来定位 Call 的思路。

网络游戏客户端和服务器端的通信基于一系列数据包，每个数据包都类似于一条指令，客户端和服务器端在这个系列指令中完成指定的动作。

相信经过之前对游戏协议的分析，读者已经比较清楚 send() 函数在客户端逻辑中的地位了。只要在 send() 函数中下断点，然后进行堆栈回溯分析，就能准确定位关键的函数调用链。在这条链上，根据上面的排除方法就能快速定位需要的 Call。

不过，send() 函数在下断点的时候，可能会碰到下面两个棘手的问题。

- 明明对 send() 函数下了断，却断不下来。
- 由于游戏中存在一个发包线程，所以既使断下 send() 函数，也无法回溯出有用的逻辑。

为什么会出现上面的问题呢？

对于第一个问题，是因为安全专家们知道 send() 函数的重要性。对于第二个问题，由于 send() 函数是客户端和服务器端通信的基础，也是暴露通信协议内容和客户端逻辑的关键，所以往往会自行加载一份包含 send() 函数的 ws2_32.dll 到内存中并隐藏它，而这将导致内存中有两份 ws2_32.dll。但是，由于游戏中的 send() 函数是自加

载的那一份，而游戏黑客们 Hook 或下断点的 send() 函数是另外一份，所以会出现即使下了断点也断不下来的情况。对定位自加载模块的介绍，请参考第 8.2.3 节。分析工具 BTAnalyse 对某款游戏枚举模块的输出信息如图 5-23 所示。

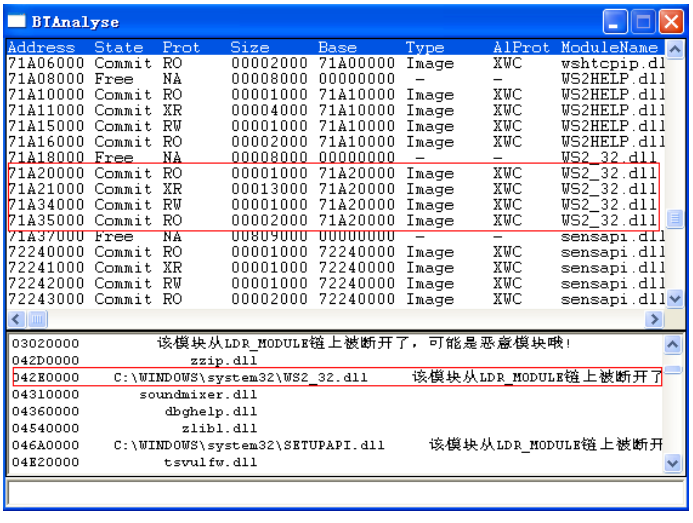


图 5-23　ws2_32.dll 枚举

可以看到，ws2_32.dll 有两份，一份的地址在 [71A20000, 7135FFF] 中，另一份的地址是从 42E0000 开始的，而且是从 LDR_MODULE 链上断开的。所以，我们有理由相信，游戏不会使用基地址是 71A20000 的这份 ws2_32.dll，而是使用基地址是 42E0000 的这份 ws2_32.dll 来实现通信。对基地址是 71A20000 的模块中的 send() 函数下断点，会发现确实断不下来。对这种情况，有以下 3 种解决方案。

- 寻找 send() 函数内调用的底层的函数，对底层函数下断点。
- 在内存中搜索 send() 函数的特征，定位被隐藏模块中的 send() 函数。
- 根据 send() 函数距离模块基地址的偏移相同来定位隐藏模块中的 send() 函数。

对于第二种方案，我们知道，send() 函数的参数里有一个是指向数据包的存放地址的，这个数据包的地址往往放在一个循环利用的大发包缓存中。所以，当我们定位这种地址的时候，需要先对发包缓存下一个条件断点，再进行堆栈回溯来定位关键 Call。

5.4　本章小结

本章主要从编写外挂的角度来阐述隐蔽 Call 和定位 Call 的相关技术。

围绕 Call 所展开的对抗始终是游戏客户端安全的重点关注方向。相信通过本章对 Call 技术各个方面详细讲解，一定能使读者对 Call 技术的理解有质的提升。

第 6 章　Hook 大全

Hook 技术在安全领域被广泛应用，是一种改变程序执行流程的技术。目前，市面上有很多关于 Hook 的书籍，其中《Rootkits ——Windows 内核的安全防护》是讲解比较深入和详细的一本。

6.1　Hook 技术简介

从代码实现的角度来看，Hook 主要分成两种：Inline Hook 和非 Inline Hook。

所谓 Inline Hook 就是将程序中某个地址处若干个字节的代码复制到其他地方，然后在原地址处写入实现跳转功能的指令（如“jmp *目标地址*”或“push *目标地址*ret”），从而改变程序的流程。下面简要描述 Inline Hook 的核心思想。

图 6-1 是对 MessageBoxA 函数进行调用的简化逻辑图。

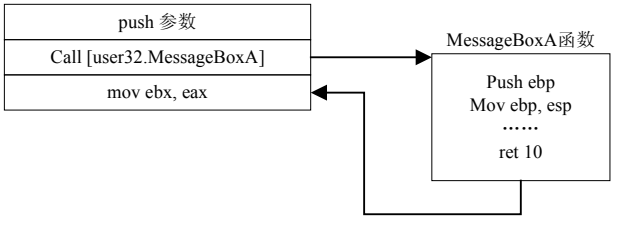


图 6-1　Inline Hook 前调用 MessageBoxA 的执行流程

图 6-2 是对 MessageBoxA 函数实现 Inline Hook 后的简化逻辑图。

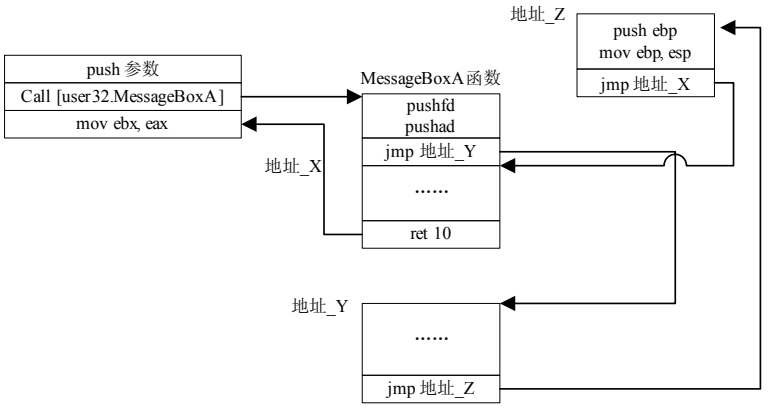


图 6-2　Inline Hook 后调用 MessageBoxA 的执行流程

通过对图 6–1 和图 6–2 的比较可知，对 MessagBoxA 函数实现 Inline Hook 后，程序对 MessageBoxA 函数的每一次调用都会被地址 _Y 处的代码截获（当然，截获后可以进行任何操作）。同时，地址 _Y 最终通过一个跳转指令跳到地址 _Z，地址 _Z 先执行 MessageBoxA 函数头的若干个指令，再通过一个跳转指令跳回 MessageBoxA 函数中的地址 _X。之后，程序的执行流程按照图 6–1 中的逻辑继续下去。

对于非 Inline Hook，我们先看一下图 6–3。

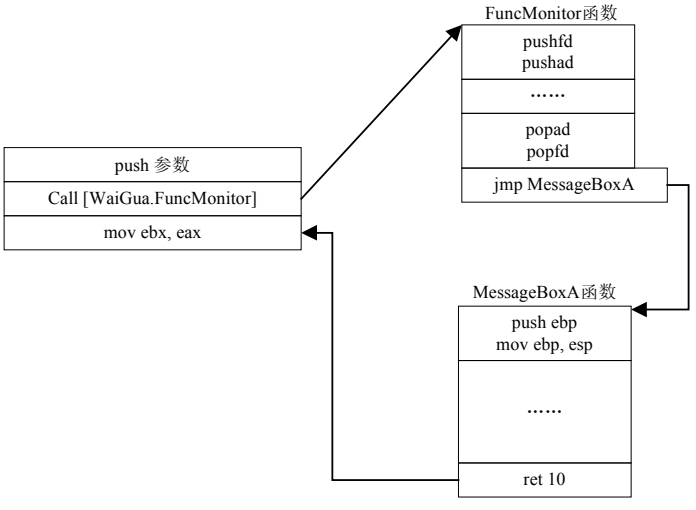


图 6-3　非 Inline Hook MessageBoxA 后的执行流程

对比图 6–1 和图 6–3 可知，这里的非 Inline Hook 只是将“call [user32.MessageBoxA]”指令改成了“call [WaiGua.FuncMonitor]”，事实上就是改变了 Call 操作的目标地址。其实，这种 Hook 方式就是第 6.2 节要介绍的 IAT Hook。

6.2 IAT Hook 在全屏加速中的应用

目前，根据加速对象的不同，加速外挂分成两种：一种是全屏加速挂，另一种是角色加速挂。全屏加速是指玩家游戏界面上所有可以移动的物体同时加速，角色加速是指游戏界面上仅玩家角色加速。

加速的好处在于能加快升级速度——被加速的玩家角色可以在游戏里灵巧地躲避怪物的攻击，并迅速攻击怪物，这无疑加快了过关的步伐。

本节讨论的加速主要针对的是改变移动速度的加速。在讨论加速之前，我们有必要了解一下游戏里的 FPS（Frames Per Second，帧频）这个概念。

FPS 即每秒帧数，它反映了每秒画面更新的数量，同时也反映了游戏主循环的更新速度。为了更好地理解目前基于 Client/Server 架构的游戏客户端中的 FPS 以及 FPS 在客户端主逻辑中的位置，下面给出游戏主逻辑架构图（如图 6–4 所示）和相应的简化主逻辑代码。

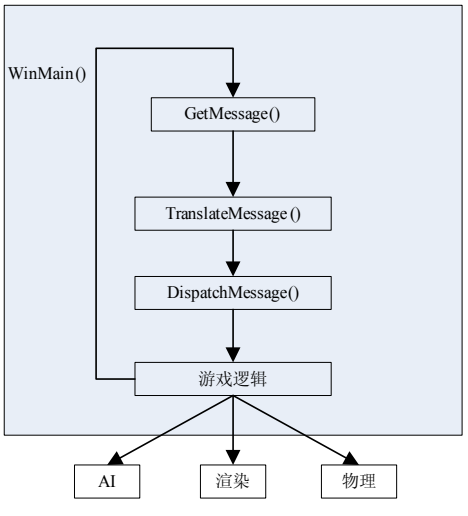


图 6-4 游戏主循环逻辑

在图 6-4 中，游戏逻辑在 1 秒内被执行的次数就是 FPS。下面给出一段简化的计算角色位移代码，使读者能更好地理解 FPS 和加速的关系。这段代码省略了包含头文件、代码规范和语法方面的细节，只从逻辑的角度来阐述问题。

```
// Game 类声明
class Game
 {
   Public:
Game();
~Game();
void Run();
   private:
DWORD m_dwPlayerInitSpeed; // 角色的初始速度，往往从脚本中读取
DWORD m_dwPlayerShowSpeed; // 角色的显示速度，即最终计算角色位移的速度
DWORD m_dwPlayerSpeedRate; // 角色速度的百分比，一般是装备等增加的百分比
DWORD m_dwPlayerCurrentPosition;        // 角色当前的位置
DWORD m_dwBeginFrameTime;  // 帧开始的时间
DWORD m_dwEndFrameTime;    // 帧结束的时间
DWORD m_dwFPS;             // FPS 值
};
// 在上面的 Game 类中，为了简化代码，暂时把角色对象的相关属性放入
// Game 构造函数进行初始化
Game::Game()
{
   m_dwFPS = 25;           // 设置帧频为每秒 25 次
m_dwPlayerCurrentPosition = 0;          // 将角色当前位置置 0
m_dwPlayerInitSpeed = 100; // 假设角色初始速度是 100
m_dwPlayerSpeedRate = 10;  // 假设角色穿上某种装备后增加 10%的速度
   m_dwBeginFrameTime = timeGetTime(); // 获取自系统启动到当前的毫秒数

}
// Run()函数负责计算角色的位置
Game::Run()
{
```

```
    DWORD dwSkipTime;           // 帧从开始到结束的时间间隔
    DWORD dwTimesPerFrame;      // 运行每帧需要的时间
    m_dwEndFrameTime  =  timeGetTime();     // 获取自系统启动到当前的毫秒数
    dwSkipTime        =  m_EndFrameTime - m_dwBeginFrameTime ;
    dwTimesPerFrame     =  1000 / m_dwFPS;     // 每帧的时间
       if( dwSkipTime > dwTimesPerFrame )
    {
           // 计算角色当前的移动速度
           m_dwPlayerShowSpeed = m_dwPlayerInitSpeed * (1 + m_dwPlayer
SpeedRate / 1000 );

           // 计算这一帧中角色的当前位移
           m_dwPlayerCurrentPosition += m_dwPlayerShowSpeed * dwSkipTime;
    }
    else
    {
           Sleep(dwTimesPerFrame - dwSkipTime);
    }

       // 更新下一帧的起始时间
       m_dwBeginFrameTime  += dwTimesPerFrame;
    }
    // 游戏主循环伪代码
    int WINAPI WinMain(
    HINSTANCE hInstance,
       HINSTANCE hPrevInstance,
       LPSTR lpCmdLine,
       int nCmdShow
    )
    {
      MSG msg;
      Game * pGame = new Game();
      while(true)
    {
       if(PeekMessage(&msg, 0 0 , PM_REMOVE))
```

```
    {
            if(WM_QUIT == msg.message)
            {
                break;
    }
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    // 计算角色的当前位移
    pGame->Run();
    }
    }
    }
```

上面的 Game::Run() 函数能够计算角色的当前位置。可以看到，其核心就是下面这段伪代码。

```
    if( dwSkipTime > dwTimesPerFrame )
    {
      // 计算角色的当前移动速度
      m_dwPlayerShowSpeed = m_dwPlayerInitSpeed * (1 + m_dwPlayerSpeed
Rate / 1000 );

      // 计算这一帧中角色的当前位移
      m_dwPlayerCurrentPosition += m_dwPlayerShowSpeed * dwSkipTime;
    }
```

对于上面这段代码，读者是否考虑过如何才能使角色实现加速呢？也就是说，在同一帧条件下，如何让角色移动得更远呢？显然，在同一段时间（假设为 T），可以通过以下两种方法使角色的位移值 m_dwPlayerCurrentPosition 变得更大。

- 增加 if 条件，判断为真的次数。
- 增加 m_dwPlayerInitSpeed、m_dwPlayerSpeedRate 或 m_dwPlayerShowSpeed。

以上第二种方法是修改角色属性中与速度相关的属性，在本节不做阐述，因为这涉及调试游戏并定位这 3 个属性在角色对象中的偏移，以及修改这 3 个属性进而实现加速的过程，与 Hook 无关。

现在，我们来分析以上第一种方法如何实现。要增加 if 条件成立的次数，必须在判断每一帧的时候增大 dwSkipTime 的值。dwSkipTIme 的值由计算式“dwSkipTime = timeGetTime() - m_dwBeginFrameTime”得出。为了增大 dwSkipTime 的值，可以 Hook 函数 timeGetTime()。每当调用该函数来获取自系统启动到当前的毫秒数时，游戏就会被截获，然后执行真正的 timeGetTime() 函数以获取一个真实值 dwTrueTime，再给这个真实值加上一个值或将其扩大若干倍后返回游戏调用的地方。这样，游戏每次调用 timeGetTime() 函数，都会得到一个比正常情况下大的值，这就间接增大了 dwSkipTime 的值。

下面给出一个笔者在分析某款外挂的过程中碰到的 Hook 游戏进程中 IAT 的 timeTimeGet() 函数从而实现全屏加速的真实例子。

游戏中正常调用 timeGetTime() 函数的逻辑如图 6-5 所示。

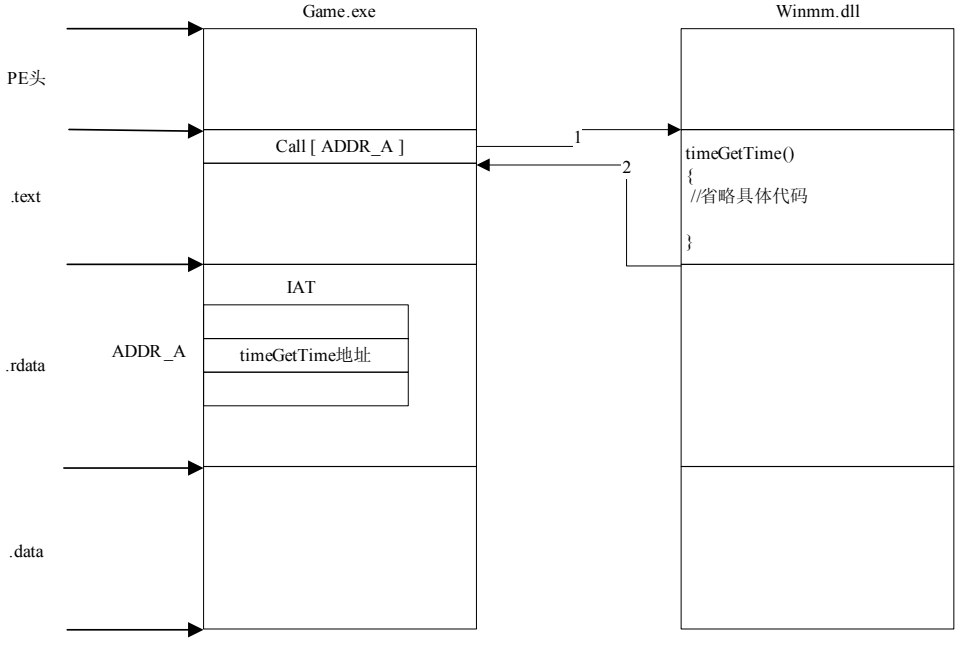


图 6-5 游戏正常调用 timeGetTime() 函数的流程

外挂模块进入 Game.exe 后，进行了一些分配和修改操作，改变了游戏调用 timeGetTime() 函数的流程，如图 6-6 所示。

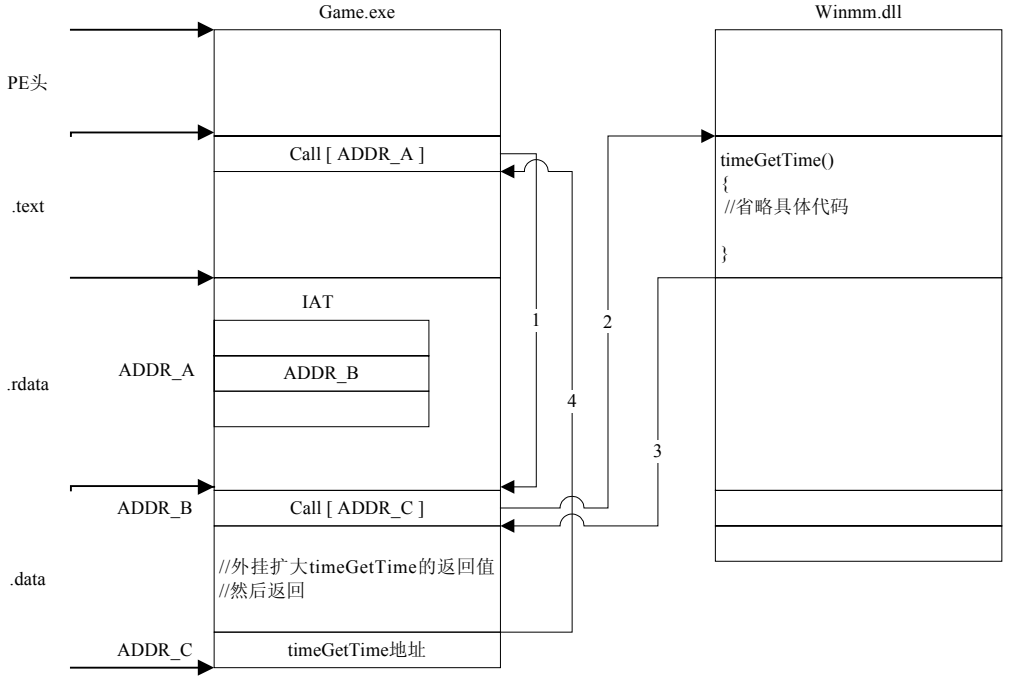


图 6-6　IAT Hook timeGetTime() 函数

在图 6–6 中，外挂在 Game.exe 的 .data 段分配了一段可读写、可执行的空间，地址 ADDR_B 处存放外挂截获 timeGetTime() 函数后的处理代码，地址 ADDR_C 处存放 timeGetTime() 函数的地址。然后，外挂把 IAT 中的“ADDR_A”修改为“ADDR_B”，这样，游戏再调用 timeGetTime() 函数时，就会转入外挂代码 ADDR_B 处开始执行。

外挂在地址 ADDR_B 处的伪代码如下。

```
DWROD Hook_TimeGetTime()
{
typedef struct  _HOOK
    {
        DWORD  dwSign;
        DWORD  dwTrueTimeGetTimeAddr;
        DWORD  dwTimeParam1;
        DWORD  dwTimeParam2;
```

```
        }HOOK, *PHOOK;
    typedef  DWORD (*PTRUE_TIMEGETTIME)();
    // 0xXXXXXXX 是外挂在.data 段存放的数据
    PHOOK pHookTime =  (PHOOK)0xXXXXXXX;
    // 获取真实的 timeGetTime()函数地址
    PTRUE_TIMEGETTIME  pTrueTimeGetTime = (PTRUE_TIMEGETTIME)pHookTime
-> dwTrueTimeGetTimeAddr;
    // 调用真实的 timeGetTime()函数，获取系统运行时间
    DWORD dwTime = pTrueTimeGetTime ();
    // dwSign 等于 0 时，使返回值加 dwTimeParam2 再返回
    if( pHookTime->dwSign == 0 )
    {
            dwTime = dwTime + pHookTime->dwTimeParam2;
            return dwTime;
    }
    // dwSign 不等于 0 时，使返回值扩大 2 倍减去 dwTimeParam1 再返回
    dwTime = dwTime - pHookTime->dwTimeParam1;
    dwTime = dwTime * 2;
        dwTime = dwTime + pHookTime->dwTimeParam1;
        return dwTime;
    }
```

看过上面的伪代码之后读者应该明白，全屏加速的关键是对 timeGetTime() 函数进行 Hook，放大系统运行时间。当然，有些游戏不一定使用 timeGetTime() 这个函数，要视情况而定，不过原理是一致的。

6.3 巧妙的虚表 Hook

虚函数是 C++ 语言中非常重要的一个概念。一个类一旦编写了虚函数，就有了虚表。虚表机制很好地保证了 C++ 语言的多态特性，而且多态特性又在游戏客户端编程中被大量运用。一个类对象的行为，如打怪、加红或者释放技能等，只能通过调用 3 种函数来实现—— 一种是类的静态函数，一种是类对象函数，一种是虚函数，而只有虚函数地址能通过类对象地址来定位。

所以，Hook 虚表中的虚函数，在分析游戏中关键内存对象的行为时具有非常重要的作用。

6.3.1 虚表的内存布局

虚表（Virtual Table）可以看成一个存放虚函数地址的一维数组，数组中每个元素占 4 字节空间。C++ 编译器会为每一个包含虚函数的类或通过继承得到的子类生成一张虚表，程序运行时，该类对象的第一个数据成员中就存放着这张虚表的地址。

为了更好地介绍对象和虚表的内存布局，下面通过两个简单的父子类定义来说明。

```
class animal
{
  public:
  animal();
  virtual void eat() = 0;
  virtual void sleep();
  virtual void breathe();
  virtual void move();
 private:
  int x;
};
```

```
class fish: public animal
{
  public:
  fish();
  virtual void eat();
  virtual void sleep();
  virtual void breathe();
  virtual void swim();
 private:
  int y;
};
```

上面这两个类的定义都涉及虚函数的声明，编译器在编译的时候，会根据声明虚函数的情况来生成对应的虚表。编译器一般把虚表放到 .rdata 段（只读数据段）。图 6–7 描述了 animal 对象和 fish 对象的内存布局和对应的虚表布局。

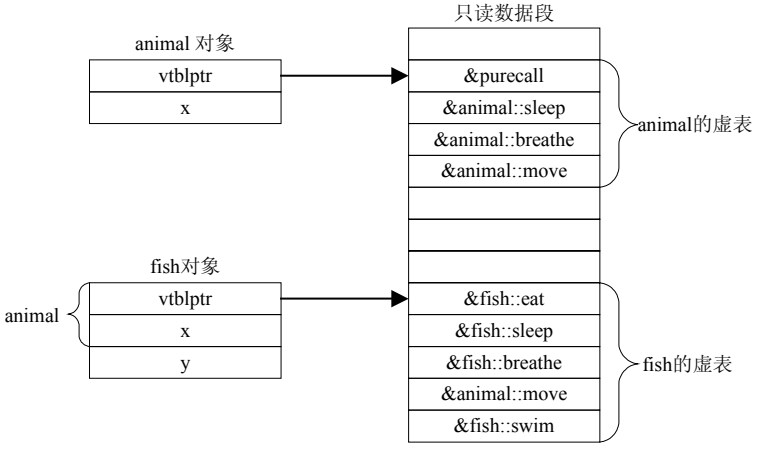


图 6-7 animal 对象和 fish 对象以及虚表的内存布局

animal 对象的纯虚函数 eat 在虚表里对应于一个 purecall。purecall 是一个异常处理函数，由编译器插入，目的是防止纯虚函数被误调用。纯虚函数没有具体实现，只在虚表里占据一个槽，以后实现这个纯虚函数时，再把虚函数地址放入 purecall 的这个槽中。

从虚表的内存布局中可以看到，它像 IAT 一样，也是一个一元数组，但我们接下来要谈的 Hook 虚表，并不是简单地替换这个数组中的某一个或某几个虚函数地址，而是设计一个监控任意类对象中所有虚函数的调用。

在对所有虚函数进行 Hook 之前，让我们先了解一下 C++ 中的 RTTI。如果不知道 RTTI 和虚表在内存布局上的依赖关系，就无法 Hook 虚表里的所有函数。

6.3.2 C++ 中的 RTTI

RTTI（Runtime Type Identification，运行时类型识别）由 C++ 编译器将对象的类型信息嵌入程序的只读数据段，以支持 C++ 的各种操作符在运行时确定（typeid）和

检查（dynamic_cast）一个对象的数据类型。关于 RTTI 的介绍，Paul Vincent Sabanal 和 Mark Vincent Yason 的论文 *Reversing C++* 推荐大家好好研究一下，网址是 http://www.blackhat.com/presentations/bh-dc-07/Sabanal_Yason/Paper/bh-dc-07-Sabanal_Yason-WP.pdf。

对逆向工作者而言，RTTI 是非常重要的，因为他们一旦定位 RTTI，就可以静态或动态地逆向出类的名字和继承关系。在 Visual C++ 6.0 中，编译器默认关闭对 RTTI 的支持，而在 Visual C++ 8.0 中，对 RTTI 的支持在默认情况下是打开的。如果程序运行时需要用到如 dynamic_cast 这样的操作符，就意味着 RTTI 是存储在二进制文件中的。对微软的编译器而言，RTTI 从虚表位置向地址减少的方向推移 4 字节的地方开始构建，具体的 RTTI 和虚表的关系还是推荐读者参考论文 *Reversing C++*。下面简单展示一下 RTTI 和虚表的关系，如图 6-8 所示。

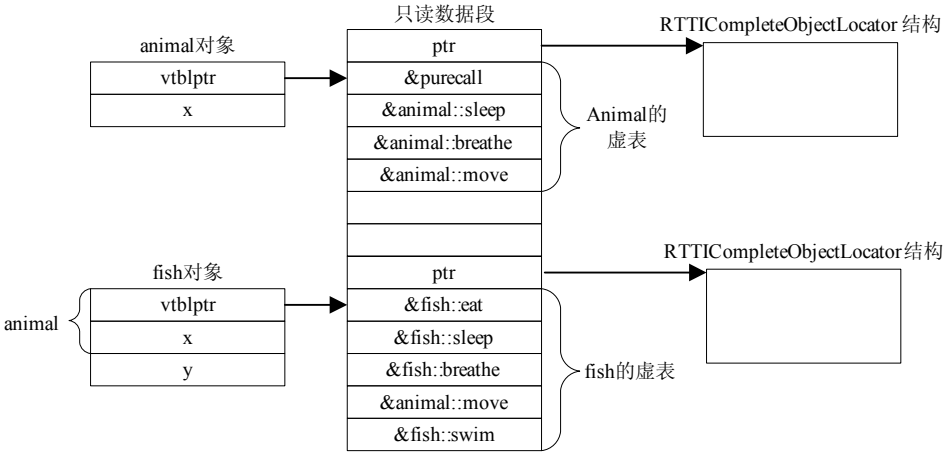


图 6-8　RTTI 和虚表的内存布局

可以看到，在虚表之前有一个指向 RTTICompleteObjectLocator 结构体的指针。关于 RTTICompleteObjectLocator 结构的具体内容，读者可以参考论文 *Reversing C++*，也可以阅读本章资源包中的示例程序代码，这里不详细讨论。

虚表之前的 4 字节是一个非常重要的指针。通过解析 RTTI，可以通过输入对象地址来输出对应类的名字和继承关系，这些信息为逆向工作者提供了非常关键的指导。

6.3.3 Hook 虚表

在第 6.3.2 节中我们已经了解，虚表其实就是存放在只读数据段中用来保存函数地址的一个一维数组。但是，由于虚表里有些位置填充的是地址 0，有些位置填充的地址是数据段的地址，所以并没有比较好的办法来获取这个数组的长度，因此，要想 Hook 虚表里所有虚函数的调用，看起来有点困难。不过，笔者已经设计并实现了一种机制来 Hook 虚表，以监控对所有虚函数的调用。

在讨论 Hook 虚表的机制之前，让我们了解一下 Hook 虚表所带来的好处。

- 从分析外挂的角度看，可以快速定位外挂模块和检测外挂所 Call 的虚表函数。
- 从分析游戏的角度看，可以快速提供与某事件相关联的关键虚函数，从而提供分析方向，不至于陷入不知所措的境地。

从汇编代码级来看，C++ 中对虚函数的调用是非常有规律的。下面是一个调用虚函数的代码片段。

```
fish MyFish;              // 定义 MyFish 对象
MyFish.sleep();           // 调用 MyFish 对象的 sleep()虚函数
```

上面对虚函数的调用，从汇编的角度来看，大致如下。

```
mov  eax, ecx             // ecx 是 MyFish 对象的地址
mov  eax, [eax]           // 获取 MyFish 对象的虚表地址
call   [eax + 0x04]       // 调用 sleep()函数
```

从图 6-8 中可以看到，sleep() 函数的地址在虚表的第 2 个槽中，所以有“call [*虚表地址* + 0x04]”。可见，虚表的调用具有固定的模式，那就是“call [*虚表地址* + *虚表偏移*]”。

如果做一个假虚表且使这个假虚表足够长，然后把假虚表的地址赋值给对象中的虚表指针，那么所有对虚函数的调用代码就会变成“call [*假虚表地址* + *虚表偏移*]”。只要假虚表的长度大于真虚表可能的最大偏移，同时使用假虚表存放监控函数的地址，那么所有对虚函数的调用都会被 Hook，这种技术暂且可以称作模糊假定。

Hook 虚表的设计图如图 6-9 所示。

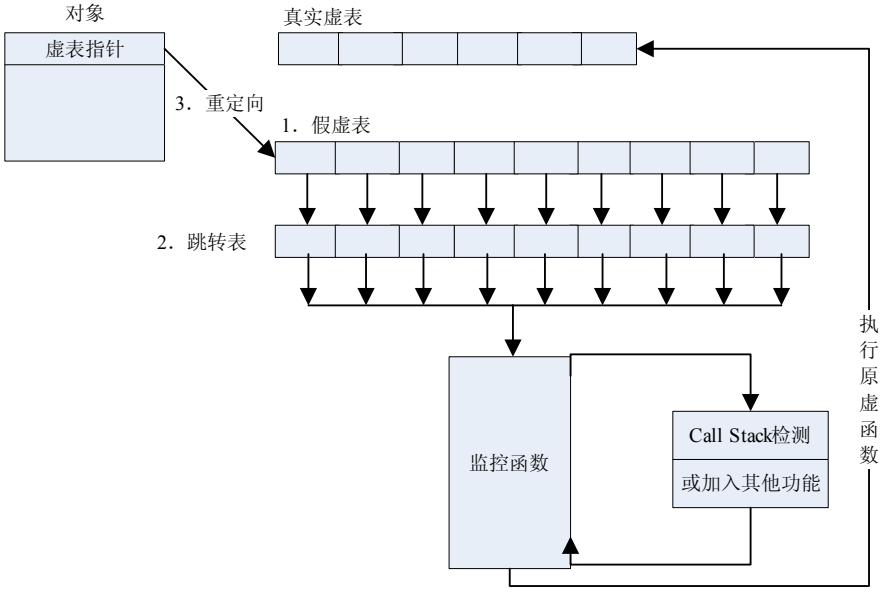


图 6-9　Hook 虚表设计图

在图 6–9 中，Hook 虚表的设计需要以下 3 个步骤。

（1）建立一张较大的假虚表。在这里，“较大的”概念是一种估算，可大可小。为了保险起见，可以申请一张有 40 000 个表项的假虚表。事实上，真实虚表中的表项绝对不可能超过这个数目，因为一个类里包含的虚函数不可能有如此大的冗余（超过 40 000 个）。建立假虚表的伪代码如下。

```
    pJmpTable = VirtualAlloc( NULL,假虚表表项的个数 * sizeof(DWORD),
MEM_COMMIT, PAGE_EXECUTE_READWRITE );
```

（2）建立跳转表并初始化跳转表和假虚表。跳转表里设置了一段代码，当程序执行对象虚函数的时候，会先执行跳转表里的代码，再由跳转表统一跳到监控函数。在监控函数里，我们可以做很多事情，其中对分析游戏和外挂比较有用的功能有 Call Stack 检测、虚函数调用监控等。跳转表项的结构示例如下。

```
    // 跳转表项的结构
    typedef struct _JMP_TABLE_ITEM
```

```
    {
        UCHAR    uPushEbx;   // push ebx (字节码等于 0x53)，用来存放真正的虚
函数地址
        UCHAR    uPushad;    // pushad (字节码等于 0x60)
        UCHAR    uPushfd;    // pushfd (字节码等于 0x9C)
        UCHAR    uMovEax[5];// mov eax, 0xffffffff (字节码等于 B8ffffffff)
        UCHAR    uPushEax;   // push eax (eax 等于虚表的编号)
        UCHAR    uJmp[5];    // jmp 0xXXXXXXXX(E9 XXXXXXXX)
        UCHAR    uSave;      // 保留，以后使用
    }JMP_TABLE_ITEM, *PJMP_TABLE_ITEM;
```

下面是一段初始化跳转表和假虚表的伪代码，完整代码在本章的资源包中。

```
    // 1、分配一个比较长的跳转表
    pJmpTable = VirtualAlloc( NULL, 跳转表项的个数 * 跳转表项的大小,
MEM_COMMIT, PAGE_EXECUTE_READWRITE );
    // 2、初始化跳转表的所有表项
    ZeroMemory( pJmpTable, 跳转表项的个数 * 跳转表项的大小 );
    for ( DWORD dwLoop = 0; dwLoop <跳转表项的个数; dwLoop++ )
    {
        pJmpTblItem->uPushEbx = 0x53;
    pJmpTblItem->uPushad  = 0x60;
    pJmpTblItem->uPushfd  = 0x9C;
    pJmpTblItem->uMovEax[0] = 0xB8;
    *((DWORD *)&(pJmpTblItem->uMovEax[1])) = dwLoop;
    pJmpTblItem->uPushEax = 0x50;
    pJmpTblItem->uJmp[0]  = 0xE9;
    // pdwMonitorFn 中存放了监控函数的地址
    *((DWORD  *)&(pJmpTblItem->uJmp[1]))  =  (DWORD)(pdwMonitorFn)  -
( (DWORD)(pJmpTblItem) + FIELD_OFFSET(JMP_TABLE_ITEM, uSave) );
    // pdwFakeVTblItem 指向假虚表的基地址
    *pdwFakeVTblItem = (DWORD)pJmpTblItem;
    pdwFakeVTblItem++;
    pJmpTblItem++;
    }
```

（3）重定向对象的虚表指针。重定向的伪代码如下。

```
// dwObjectBaseAddr 是对象基地址，pdwFakeVTblItem 是假虚表基地址
*(PDWORD)dwObjectBaseAddr = pdwFakeVTblItem;
```

本节资源包中给出的真实监控虚表的代码比以上伪代码要复杂得多，其中专门开辟了一个线程进行替换操作。在替换虚表地址之前，还进行了悬挂游戏主线程的操作。

6.4 Detours Hook

本节将介绍 Detours Hook 的原理、Detours 库函数的用法以及如何设计一个基于 Detours 库接口的动态 Hook 引擎。

6.4.1 Detours 简介

Detours 是微软开发的一个函数库，主要用于动态 Hook 运行中的程序，其具体介绍参见 http://research.microsoft.com/en-us/projects/detours/。

在游戏或外挂分析过程中，可以利用 Detours 库提供的接口来动态 Hook 任意地址，截获函数调用，输出打印信息。

如果要使用 Detours 库提供的接口，首先要使程序包含 detours.h 文件，以及导入 detoured.lib 文件和 detours.lib 文件，示例如下。

```
// 使程序包含 detours.h 文件
#include "../3rdParty/detours/include/detours.h"
// 导入 detoured.lib
#pragma comment(lib, "../3rdParty/detours/lib/detoured.lib")
// 导入 detours.lib
#pragma comment(lib, "../3rdParty/detours/lib/detours.lib")
```

6.4.2 Detours Hook 的 3 个关键概念

要理解 Detours Hook，必须理解 Detours 中的如下 3 个关键概念。

- Target 函数：即要 Hook 的目标函数或目标地址。

- Trampoline 函数：即跳板函数，主要负责保存原始 Target 函数头的若干条指令，并加上一个跳转指令以保持对原始 Target 函数调用的语义完整性。
- Detour 函数：即截获 Target 函数的调用之后所要执行的自定义函数。

在 Detours Hook 中，Trampoline 函数与 Target 函数之间的关系如图 6-10 所示。

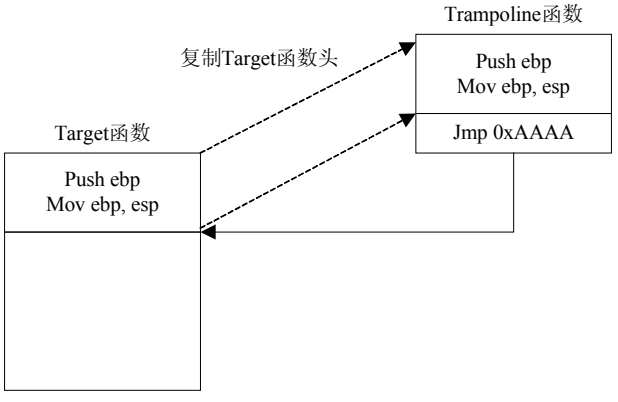


图 6-10 Trampoline 函数与 Target 函数的关系

可以看出，Trampoline 函数是由 Target 函数头加 jmp 指令组成的。

Target 函数、Detour 函数以及 Trampoline 函数之间的关系如图 6-11 所示。

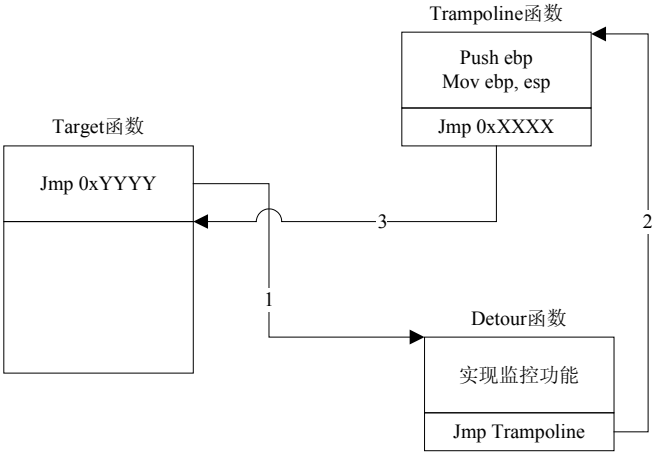


图 6-11 Target 函数、Detour 函数和 Trampoline 函数的关系

从图 6-11 中可知，一旦 Target 函数被执行，程序将按照 1→2→3，即 Target → Detour → Trampoline 的顺序执行，最后回到 Target 函数的执行过程。

6.4.3 Detours Hook 的核心接口

Detours Hook 的核心接口主要负责实现 Hook 和 Unhook 功能，具体如下。

- DetourTransactionBegin(VOID)：开始 Hook 或 Unhook 当前事务。引入事务机制的目的是保证操作的原子性。
- DetourUpdateThread(DWORD dwThreadID)：对进程中每个可能调用 Target 函数的线程，都要使用 DetourUpdateThread 将其加入 update 队列。这是因为 Hook 时只修改 Target 函数的前几字节，如果某个线程刚好执行到这几字节，粗暴的修改会造成该线程出现异常。因此，Detours 进行事务处理时，会先枚举并暂停 update 队列中的所有线程，以获取它们的指令指针。如果发现这种情况，就将指令指针修改到 Trampoline 函数上，从而避免程序崩溃。
- DetourAttach(PVOID *ppPointer, PVOID pDetour)：DetourAttack 函数负责建立图 6-11 中的 Target 函数、Detour 函数和 Trampoline 函数之间的跳转关系。其中，ppPointer 参数返回指向 Trampoline 函数的地址，而 pDetour 参数传入待跳转的 Detour 函数的地址。
- DetourTransactionCommit(VOID)：提交当前事务，实现真正的 Hook。
- DetourDetach(PVOID *ppPointer, PVOID pDetour)：DetourDetach 与 DetourAttack 刚好相反，用于解除图 6-11 中 3 个函数之间的跳转关系，即 Unhook。其中，ppPointer 是 Trampoline 函数的地址，pDetour 是 DetourAttack 中指定的 Detour 函数的地址。

通过上面介绍的 5 个核心接口，经过简单的逻辑，就可以实现 Detours Hook 和 Unhook 了。

下面是一个 MessageBox 函数实现 Detours Hook 和 Unhook 的示例。

```
// 声明接收指向 Trampoline 函数地址的指针
PVOID pTrampoline = NULL;
// 声明指针 pfnMessageBoxA 为指向 MessageBoxA 函数的函数指针
```

```
    typedef int (*pfnMessageBoxA)( HWND hWnd, LPCSTR lpText, LPCSTR
lpCaption, UINT uType);
    // 给指针 pfnMessageBoxA 赋值
    pfnMessageBoxA  =  GetProcAddress(GetModuleHandle("user32.dll"),
"MessageBoxA");

    // Detours Hook
    BOOL DetourHook(VOID)
    {
        // 开始 Hook
        DetourTransactionBegin();

        // 只有一个线程，所以 GetCurrentThread
        DetourUpdateThread( GetCurrentThread());

        // 添加 MessageBoxA 的 Hook
        if( DetourAttach( &(PVOID&) pTrampoline, pfnMessageBoxA) !=
NO_ERROR)
        {
            printf( "Hook MessageBoxA fail.\n");
        }

        // 完成事务
        if( DetourTransactionCommit() != NO_ERROR)
        {
            printf( "DetourTransactionCommit fail\n");
            return FALSE;
        }
        else
        {
            printf( "DetourTransactionCommit ok\n");
            return TRUE;
        }
    }
```

```
    // Detours Unhook
    BOOL DetourUnhook(VOID)
    {
        // 开始 Unhook
        DetourTransactionBegin();

        // 只有一个线程，所以 GetCurrentThread
        DetourUpdateThread( GetCurrentThread());

        // Unhook
        if(DetourDetach ( &(PVOID&) pTrampoline, pfnMessageBoxA) != NO_
ERROR)
        {
            printf( "UnHook MessageBoxA fail.\n");
        }

        // 完成事务
        if( DetourTransactionCommit() != NO_ERROR)
        {
            printf( "DetourTransactionCommit fail\n");
            return FALSE;
        }
        else
        {
            printf( "DetourTransactionCommit ok\n");
            return TRUE;
        }
    }
```

6.4.4 Detours Hook 引擎

Detours Hook 引擎采用上面介绍的 Detours Hook 机制，经过精心的设计，能够支持动态 Hook 几乎任意地址以方便管理，而不用为了 Hook 一个地址去增加代码和重新编译代码（注意：这里的“任意地址”在可以被修改的地址区域内）。

首先还是让我们看看如图 6–12 所示的这个引擎的设计概要，然后再详细分析每一块的具体内容。

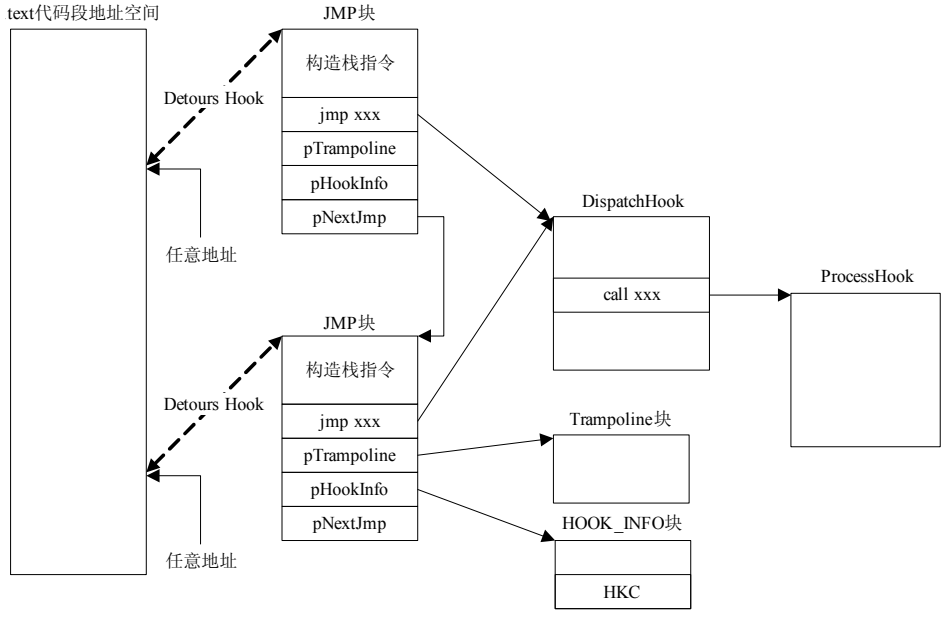


图 6-12 Detours Hook 引擎设计概要

如图 6–12 所示的 Detours Hook 引擎，从涉及的内存结构上看，主要由 4 个块组成，分别是 JMP 块、HOOK_INFO 块、Trampoline 块和 HKC 块，而从处理函数上看，主要由 DispatchHook 和 ProcessHook 组成。下面从每个块的组成以及相互间的关系说起，然后介绍涉及的两个核心函数。

1. HKC 块

HKC 块的结构体定义如下。

```
// 导出供用户使用的数据结构
typedef struct _HOOK_CALLBACK
{

    DWORD  dwHookAddr;
```

```
    DWORD  dwCallBackAddr;
    // 0 表示禁用，1 表示启用
    DWORD              dwEnable;
    // 保持互斥访问 dwEnable
    LONG               lMutexAcc;
    // 被 Hook 的函数的名字
    TCHAR              FuncName[FUNC_NAME_LENGTH];
    // 被 Hook 的函数所在 DLL 的名字
    TCHAR              DllName[DLL_NAME_LENGTH];
    // 堆栈回溯的层数
    DWORD                dwCallLevel;
    // 供用户使用的自定义指针
    PVOID              pUserDefine;
}HOOK_CALLBACK,*PHOOK_CALLBACK,HKC;
```

从 HKC 结构体的定义来看，这个数据结构是在 Hook 之前供用户输入配置参数的，例如登记被 Hook 函数的名字、出自哪个模块、在堆栈回溯的时候要回溯多少层以及开启或禁用该 Hook 等信息。

2. HOOK_INFO 块

HOOK_INFO 块结构体的构成如下。

```
// Hook 的输入信息
typedef struct _HOOK_INFO
{
    DWORD              dwEax;
    DWORD              dwEcx;
    DWORD              dwEdx;
    DWORD              dwEbx;
    DWORD              dwEsp;
    DWORD              dwEbp;
    DWORD              dwEsi;
    DWORD              dwEdi;
    DWORD              dwHookID;
```

```
    // “true”表示 Unhook，否则表示 Hook
    BOOL              bUnHook;
    HOOK_CALLBACK      stHookCallback;
}HOOK_INFO, *PHOOK_INFO;
```

HOOK_INFO 结构体保存了与此次 Hook 相关的很多有用的信息，主要分成两部分：一部分是由用户自己填写的信息，这些信息统一登记在 stHookCallback 结构体中；另一部分是 Hook 后由程序填写和使用的信息，如 Hook ID、通用寄存器信息等。

3. JMP 块

JMP 块的职责是精心构造一个运行时栈，以连接 HOOK_INFO 等块，具体的数据结构如下。

```
typedef struct _JMP_BLOCK
{

    UCHAR    uPushEax;       // push eax 0x50
    UCHAR    uPushad;        // pushad 0x60
    UCHAR    uPushfd;        // pushfd 0x9C
    UCHAR    uMovEax[5];     // mov eax, 0xffffffff = B8 ffffffff
    UCHAR    uPushEax2;      // push eax = 0x50
    UCHAR    uJmp[5];        // jmp 0xfffffffe = e9...
    PVOID    pOldFuncAddr;   // Detour 后的 Trampline 地址
    PHOOK_INFO   pHookInfo; // 被 Hook 的函数的参数列表信息
    struct _JMP_BLOCK  *pNext;

}JMP_BLOCK, *PJMP_BLOCK,JMP,*PJMP;
```

JMP 块的设计至关重要，因为由图 6-12 可知，JMP 块相当于被 Hook 的任意地址与 Detour 块之间的桥梁，起着转移 EIP 和保存执行环境的作用。

4. Trampoline 块

Trampoline 块是 DetourAttach 成功以后由 Detour 生成的，我们只需要把这个地址记录到 JMP 块中，以便继续执行原始函数。

到目前为止，相信读者已经了解了设计一个 Detour 动态 Hook 引擎涉及的 4 个核心数据结构。那么，下面就让我们看看一个简化的通用 Hook 引擎函数是什么样的，示例如下。

```
    BOOL DetourHook(PHOOK_CALLBACK pstHookBack)
    {
            bool bSuccess = false;
            FPROCESSFUNC pProcFun = NULL;
            PVOID  pNewAddr = NULL;
            PJMP_BLOCK   pJmp = NULL;
            PVOID  pMnFn = NULL;

        __try
        {

            // 创建 JMP 结构
            pJmp = CreateJMPBlock();
            if ( !pJmp )
            {
                MyDbgPrint("[-]%s CreateJMPBlock error!\n", FLINFO);
                return bSuccess;
            }
            // 全局变量 dwHookId2，记录目前有多少个 Hook
            g_dwHookId2++;
            pJmp->pHookInfo->dwHookID    = g_dwHookId2;
            // 初始化的时候都是 Hook 状态
            pJmp->pHookInfo->bUnHook     = false;
            // 复制用户登记的 HKC 信息
            memcpy(&pJmp->pHookInfo->stHookCallback,pstHookBack,sizeof
(HOOK_CALLBACK));

            // 实施 Detour
            pNewAddr = (PVOID)pJmp;
// StartDetourHook()函数后面就会给出，主要是调用了 Detours Hook 的核心接口
```

```
            pMnFn = StartDetourHook( (PVOID)(pstHookBack->dwHookAddr),
pNewAddr, pJmp);

            // 保存被 Hook 函数的 Trampoline 地址
            if ( !pMnFn )
            {
                if ( pJmp )
                {
                    if ( pJmp->pHookInfo )
                    {
                        MMFree(pJmp->pHookInfo);
                    }
                    MMFree(pJmp);
                }
                return bSuccess;
            }
            // pMnFn 很关键，就是 Trampoline
            pJmp->pOldFuncAddr = pMnFn;
            //如果成功，加入全局 JMP 链表
            pJmp->pNext = g_pJmpBlockHeader;
            g_pJmpBlockHeader   = pJmp;

        }
        __except( EXCEPTION_EXECUTE_HANDLER )
        {
            MyDbgPrint("[!] DetourHook Exception Exit!\n");
            return bSuccess;
        }

        return bSuccess;
    }
```

DetourHook() 函数看上去并不复杂，而且逻辑清晰，其中包含两个关键函数，一个是 CreateJMPBlock，另一个是 StartDetourHook。下面就让我们分别看看这两个函数的实现。

CreateJMPBlock() 函数的实现示例如下。

```
PJMP_BLOCK CreateJMPBlock()
{
    PJMP_BLOCK          pAllocJmp    = NULL;
    PHOOK_INFO          pAllocFnInfo = NULL;
    ULONG               uLoop;
    PLONG               puMonitorFn;
    // 声明接收函数地址指针变量
    // 传递分发跳转的函数地址 DispatchHook
    typedef VOID (*pMonitorFn)(PVOID);
    pMonitorFn pMonFn;
    // DispatchHook 函数很关键，后面会专门介绍
    pMonFn        =  DispatchHook;
    puMonitorFn   =  (PLONG)pMonFn;
    // 在堆上分配 JMP_BLOCK 块
    pAllocJmp   =   (PJMP_BLOCK)MMalloc( sizeof(JMP_BLOCK) );
    if ( !pAllocJmp )
    {
        OutputDbgInfo(("[-]%s MMalloc fail !\n", FLINFO));
        return NULL;
    }
    DWORD dwOlpPageProtect;
        // 将 JMP_BLOCK 块设置为可读写执行权限
        BOOL bSuccess  =  VirtualProtect(pAllocJmp,
sizeof(JMP_BLOCK),
PAGE_EXECUTE_READWRITE,
&dwOlpPageProtect);
    if (!bSuccess)
    {
        MyDbgPrint("CreateJMPBlock VirtualProtect Error!");
        if (pAllocJmp)
        {
            MMFree(pAllocJmp);
        }
```

```
            return NULL;
        }
        // 在堆上分配 HOOK_INFO 块
        pAllocFnInfo    =  (PHOOK_INFO)MMalloc( sizeof(HOOK_INFO));

        if ( !pAllocFnInfo )
        {
            MyDbgPrint("[-]%s MMalloc HOOK_INFO fail !\n", FLINFO);
            if ( pAllocJmp )
            {
                MMFree( pAllocJmp );
            }
            return NULL;
        }
        ZeroMemory(pAllocFnInfo, sizeof(HOOK_INFO));

        // 默认情况下，启用该点的功能
        pAllocJmp->pHookInfo       =   pAllocFnInfo;

        // 指令配置
        //push eax 只是为了占据一个栈空间为后面保存的地址所用
        pAllocJmp->uPushEax =    0x50;
    // pushad
    pAllocJmp->uPushad     =    0x60;
        // pushfd
        pAllocJmp->uPushfd  =    0x9C;
        // mov eax, 0xXXXXXXXX
        pAllocJmp->uMovEax[0]   =    0xB8;
    *((LONG *)&(pAllocJmp->uMovEax[1])) = (LONG)(pAllocJmp) + FIELD_
OFFSET(JMP_BLOCK, pOldFuncAddr);
        // push eax
        pAllocJmp->uPushEax2        =  0x50;
        // jmp MonitorFunc, MonitorFunc 的地址是 DispatchHook
        pAllocJmp->uJmp[0]      =  0xE9;
```

```
    *((LONG*)&(pAllocJmp->uJmp[1]))    =    (LONG)(puMonitorFn)      -
( (LONG)(pAllocJmp) + FIELD_OFFSET(JMP_BLOCK, pOldFuncAddr));
       FlushInstructionCache(     GetCurrentProcess(),     pAllocJmp,
sizeof(JMP_BLOCK));
       return pAllocJmp;
    }
```

CreateJMPBlock() 函数创建 JMP 块之后，需要将要 Hook 的地址和 JMP 地址通过 StartDetourHook() 函数联系起来。

下面再让我们看看 StartDetourHook() 函数的实现。

```
    // StartDetourHook()函数调用 Detour 接口实施真正的 Hook
    PVOID StartDetourHook(
                        PVOID pTargetFnAddr, // 被 Hook 函数的地址
                        PVOID pNewFnAddr,    // 新函数的地址,也是 JMP 地址
                        PJMP_BLOCK  pJmp     // JMP 块的地址
                        )
    {
       PVOID        pTrampoline = NULL;

       // 准备开始 Detour
       DetourTransactionBegin();

       // 暂停除了当前线程外的其他线程
       DetourUpdateThread(GetCurrentThread());

       // 开始 Detour
       typedef VOID (WINAPI *pFn)(PVOID);
       pFn pOldFn = (pFn)pTargetFnAddr;
       pFn pNewFn = (pFn)pNewFnAddr;
       if( DetourAttach( &(PVOID&)pOldFn, pNewFn) != NO_ERROR)
       {
           MyDbgPrint("[-]%s DetourAttach error!\n", FLINFO);
           return NULL;
       }
```

```
    // 提交当前事务，实现真正的 Hook
    if( DetourTransactionCommit() != NO_ERROR)
    {
        MyDbgPrint("[-]%s DetourTransactionCommit error!\n", FLINFO);
        return NULL;
    }

    pTrampoline = (PVOID)pOldFn;
    // pTrampoline 非常关键
    pJmp->pOldFuncAddr = pTrampoline;

    return pTrampoline;
}
// 分配一个 HOOK_INFO 块并初始化
pHookInfo = new HOOK_INFO;
if( !pHookInfo )
{
   printf("new HOOK_INFO fail!");
   return FALSE;
}
```

经过上面的函数调用之后，一个完整的 Detours Hook 引擎就完成了，但是还需要构建两个关键函数，一个是 DispatchHook，一个是 ProcessHook，示例如下。

```
// DispatchHook()函数是用纯汇编语言编写的一个裸函数
VOID  __declspec(naked) DispatchHook(PVOID pFirstParamOffset)
{

    __asm
    {
        // 保存 trampoline 地址
        mov eax, dword ptr [esp]
        mov eax, dword ptr [eax]
        mov dword ptr [esp+40],eax
    }
```

```
__asm
{
    // 保存调用环境的寄存器值
    mov eax, dword ptr [esp] //eax == offset pOldFunc

    mov ebx, dword ptr [esp+36]
    mov eax, dword ptr [eax+4]     // eax == pHookInfo
    mov dword ptr [eax], ebx       // save eax

    mov ebx, dword ptr [esp+32]
    mov dword ptr [eax+4], ebx     // save ecx

    mov ebx, dword ptr [esp+28]    // save edx
    mov dword ptr [eax+8], ebx

    mov ebx, dword ptr [esp+24]    // save ebx
    mov dword ptr [eax+12], ebx

    mov ebx, dword ptr [esp+8]
    mov dword ptr [eax+28], ebx    // save edi

    mov ebx, dword ptr [esp+12]
    mov dword ptr [eax+24],ebx     // save esi

    mov ebx, dword ptr [esp+20]    // save esp
    lea ebx, dword ptr [ebx+4]
    mov dword ptr [eax+16], ebx

    mov ebx, dword ptr [esp+16]
    mov dword ptr [eax+20],ebx      //save ebp
}

__asm
{
    // 调用 ProcessHook 进行真正的分发处理
```

```
        mov eax, dword ptr [esp]
        push eax
        call  DWORD PTR [g_pProcessFn]
    }

    __asm
    {

        cmp eax, -1
        jnz  _NO_EXCUTE_HOOK_FUNC

        // 执行原始函数
        __asm
        {
            // 恢复寄存器
            mov eax, dword ptr [esp] //eax == offset pOldFunc

            mov eax, dword ptr [eax+4]      // eax == pHookInfo
            mov ebx, dword ptr [eax]        // resume eax
            mov dword ptr [esp+36], ebx

            mov ebx,dword ptr [eax+4]       // resume ecx
            mov dword ptr [esp+32],ebx

            mov ebx, dword ptr [eax+8]      // resume edx
            mov dword ptr [esp+28],ebx

            mov ebx, dword ptr [eax+12]     // resume ebx
            mov dword ptr [esp+24], ebx

            mov ebx, dword ptr [eax+28]
            mov dword ptr [esp+8], ebx      // resume edi

            mov ebx, dword ptr [eax+24]
```

```
            mov dword ptr [esp+12],ebx      // resume esi
        }

        pop eax
        popfd
        popad
        ret
// 不执行被 HOOK 的函数，直接返回
_NO_EXCUTE_HOOK_FUNC:

        mov [esp+40], eax

            __asm
        {
            // 恢复寄存器
            mov eax, dword ptr [esp]        //eax == offset pOldFunc

            mov eax, dword ptr [eax+4]      // eax == pHookInfo
            mov ebx, dword ptr [eax]        // resume eax
            mov dword ptr [esp+36], ebx

            mov ebx,dword ptr [eax+4]       // resume ecx
            mov dword ptr [esp+32],ebx

            mov ebx, dword ptr [eax+8]      // resume edx
            mov dword ptr [esp+28],ebx

            mov ebx, dword ptr [eax+12]     // resume ebx
            mov dword ptr [esp+24], ebx

            mov ebx, dword ptr [eax+28]     // resume edi
            mov dword ptr [esp+8],ebx

            mov ebx, dword ptr [eax+24]
```

```
            mov dword ptr [esp+12],ebx      // resume esi
        }

        pop eax
        popfd
        popad
        mov eax, [esp]
        lea esp,[esp+4]
        // eax 是原始函数参数的个数
        // 因为是动态和任意 Hook，所以应采用枚举来配对参数个数，进行堆栈平衡
        cmp eax,0
        jz  _PARAM_0_
        cmp eax,1
        jz  _PARAM_1_


_PARAM_0_:
        ret

_PARAM_1_:
        ret 4
}
}
```

接下来让我们看看最后处理 Hook 的 ProcessHook() 函数，示例如下。

```
DWORD _stdcall ProcessHook(PVOID pFirstParamOffset)
{

    PHOOK_INFO pHookInfo = NULL;
    DWORD      dwCallAddr    = 0;
    DWORD      dwFuncParam[16]   = {0};
    DWORD      dwLoop        = 0;
    DWORD      dwRet;
    FPROCESSFUNC pProcessFun = NULL;
```

```
        pHookInfo = *(PHOOK_INFO*)((PBYTE)pFirstParamOffset + 4);

        if (pHookInfo->stHookCallback.dwEnable == 1)
        {
            // 分发给各个点的处理函数
            pProcessFun   =   (FPROCESSFUNC)pHookInfo->stHookCallback.
dwCallBackAddr;
            dwRet = pProcessFun((PVOID)pHookInfo);
        }
        else
        {
            dwRet = -1;
        }
        return dwRet;
    }
```

图 6-13 展示了一个函数在 Detours Hook 引擎中的执行逻辑，详细且形象地说明了 EIP 由函数跳入 JMP 块，再跳入 Detour 块时栈的情况。

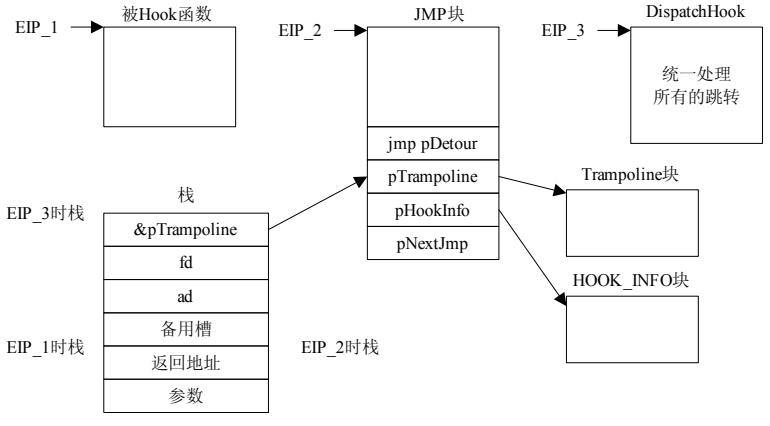


图 6-13　Detours Hook 后的运行时栈

在图 6-13 中，当 EIP 从 EIP_1 跳入 EIP_2，最后跳到 EIP_3 的时候，根据之前 JMP 块的设计，此时栈中的 [esp] 是 Trampoline 地址，而 [esp+4] 是 pHookInfo 变量的地址，所以，在 ProcessHook() 函数中，通过“PHOOK_INFO pHookInfo = *(PHOOK_

INFO*) ((PBYTE)pJmpOffse+4)”语句就可以获取此次 Hook 的相关信息。同时，根据当时栈的情况，可以直接返回 Trampoline，这样既不会改变原始程序的执行路径，也可以获取执行环境信息。

6.5 高级 Hook

本节将介绍硬件断点 Hook。因为这种 Hook 与 Windows 异常机制紧密相关，所以，必须先向读者简单介绍 Windows 的 S.E.H（Struct Exception Handler，结构化异常处理）机制和 V.E.H（Vector Exception Handler，向量异常处理）机制。

6.5.1 S.E.H 简介

S.E.H 是操作系统提供给线程来感知和处理异常的一种回调机制。S.E.H 在线程栈上以单链表的形式存在，如图 6-14 所示。

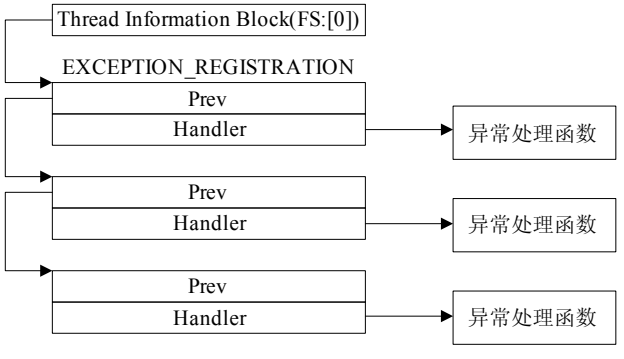


图 6-14 S.E.H 链表

在 Intel Win32 平台上，由于 FS 寄存器总是指向当前的 TIB（线程信息块），因此在 FS:[0] 处能找到最近的一个 EXCEPTION_REGISTRATION 结构。

当我们通过 try/catch（C++）或 __try/__except（微软 C++ 编译器支持）等操作来注册 S.E.H 的时候，FS:[0] 会指向新的 S.E.H，且新的 S.E.H 的 Prev 字段会指向之前 FS:[0] 指向的 S.E.H，整个操作类似于单链表的表头插入操作。

S.E.H 中关于 EXCEPTION_REGISTRATION 的定义示例如下。

```
typedef struct EXCEPTION_REGISTRATION
{
    struct EXCEPTION_REGISTRATION * prev ;
    _except_handler  handler ;
}  EXCEPTION_REGISTRATION,  PEXCEPTION_REGISTRATION
```

S.E.H 的异常处理函数在 EXCPT.H 中的原型示例如下。

```
EXCEPTION_DISPOSITION
 __cdecl _except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext
    );
```

_except_handler 的第一个参数 EXCEPTION_RECORD 在 WINNT.H 中的定义示例如下。

```
typedef struct _EXCEPTION_RECORD {
 DWORD ExceptionCode;
 DWORD ExceptionFlags;
 struct _EXCEPTION_RECORD *ExceptionRecord;
 PVOID ExceptionAddress;
 DWORD NumberParameters;
 DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
 }  EXCEPTION_RECORD;
```

对 S.E.H 的详细介绍，读者可以参考文档 http://www.microsoft.com/msj/0197/exception/exception.aspx 和《0day 安全：软件漏洞分析技术》一书，这里就不赘述了。

6.5.2 V.E.H 简介

从 Windows XP 开始，微软在原有的基于线程的 S.E.H 异常处理的基础上，增加了基于进程的 V.E.H 异常处理。

对于 V.E.H，读者可以先了解下面这些信息。

- V.E.H 和 S.E.H 不同，前者是基于进程的，而后者是基于线程的，且 Windows 提供了 API 注册回调函数。
- V.E.H 处理异常的优先级低于调试器，高于 S.E.H，即 KiUserException Dispatcher() 函数首先检查进程是否处于被调试状态，然后检查 V.E.H 链表，最后才是检查 S.E.H 链表。
- V.E.H 以双链表的形式保存在堆中，且 V.E.H 的节点可以挂接在双链表的头部或尾部。

Windows 提供的注册 V.E.H 的回调 API 示例如下。

```
PVOID AddVectoredExceptionHandler(
ULONG FirstHandler,
PVECTORED_EXCEPTION_HANDLE VectoredHandler
);
```

- FirstHandler：如果大于 0 则从头部插入，否则从尾部插入。
- VectourHandler：异常处理函数的地址。

根据 MSDN 提供的信息，V.E.H 节点结构示例如下。

```
typedef struct _VECTORED_EXCEPTION_NODE {
   LIST_ENTRY ListEntry;
   PVECTORED_EXCEPTION_HANDLER    pfnHandler;     // 被加密的指针
   } VECTORED_EXCEPTION_NODE, *PVECTORED_EXCEPTION_NODE;
```

通过对 V.E.H 节点结构的了解，我们大致可以知道 V.E.H 链表的形式，如图 6-15 所示。

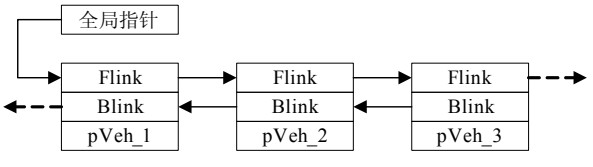


图 6-15 V.E.H 链表结构

在外挂与反外挂的对抗中，如果要枚举 V.E.H 链表，一种方案是获取 Windows

中 V.E.H 全局指针的值来遍历 V.E.H 双链表。另外提供一种比较巧妙的思路，这种思路在安全领域的其他场合也是有用的，那就是调用 AddVectored ExceptionHandler()函数注册一个假的 V.E.H 节点，从函数的返回值中获取链表的头指针或尾指针，从而方便地遍历整个 V.E.H 双链表，最后调用 RemoveVectoredExceptionHandler() 函数删除这个假的 V.E.H 节点，我们可以把这种技术称为虚假注册。关于 V.E.H 的详细介绍参见文档 http://msdn.microsoft.com/en-us/magazine/cc301714.aspx。

6.5.3 硬件断点

IA-32 处理器定义了 8 个调试寄存器，即 DR0 ~ DR7。DR0 ~ DR3 用来指定断点的内存地址或 I/O 地址；DR4 和 DR5 是保留的；DR6 的作用是当调试事件发生时向调试器报告事件的详细信息，以供调试器判断发生的是何种事件；DR7 用来进一步定义中断条件。对于硬件断点和调试器寄存器的详细信息，读者可以参考张银奎老师的《软件调试》一书或《Intel CPU 体系架构手册》。

硬件断点 Hook 是结合 DR0 ~ DR3 调试寄存器和 Windows S.E.H 或 V.E.H 机制所引入的一种 Hook 机制。在很多游戏反外挂系统中，都有一套校验游戏代码完整性的机制，所以如果 Hook 采用的方式是修改代码，那么很容易就会被检测到。然而，硬件断点 Hook 并不涉及修改代码，所以它的优点主要体现在隐蔽性上。

在进行硬件断点 Hook 之前，让我们先了解一下线程的上下文环境以及 Windows 下线程与 CPU 之间的关系。

MSDN 的资料显示，线程的执行环境 CONTEXT 的结构体示例如下。

```
 typedef struct _CONTEXT
{
        DWORD   ContextFlags;
        DWORD   Dr0;
        DWORD   Dr1;
        DWORD   Dr2;
        DWORD   Dr3;
        DWORD   Dr6;
        DWORD   Dr7;
        FLOATING_SAVE_AREA FloatSave;
```

```
        DWORD   SegGs;
        DWORD   SegFs;
        DWORD   SegEs;
        DWORD   SegDs;
        DWORD   Edi;
        DWORD   Esi;
        DWORD   Ebx;
        DWORD   Edx;
        DWORD   Ecx;
        DWORD   Eax;
        DWORD   Ebp;
        DWORD   Eip;
        DWORD   SegCs;
        DWORD   EFlags;
        DWORD   Esp;
        DWORD   SegSs;
} CONTEXT;
```

从 CONTEXT 的结构来看，线程执行环境主要是以寄存器为主，其中有通用寄存器、硬件断点寄存器、EIP（指令指针）等。

从功能部件上来看，CPU 由运算器、寄存器、控制器、系统总线接口等组成，如图 6–16 所示。

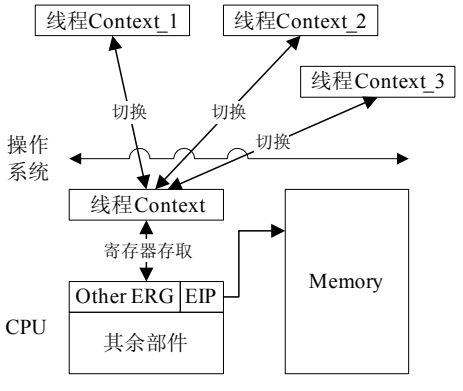


图 6-16 CPU 与线程简化关系图

众所周知，Windows 是基于线程调用的多任务抢占式操作系统。根据图 6–16，假设目前 CPU 运行的是线程 1，这时优先级更高的线程 2 欲抢占执行，那么，操作系统首先会把线程 1 从当时的执行环境（即 CPU 的寄存器信息）中取出，放入线程 1 的 CONTEXT 结构中（即图 6–16 中的线程 Context_1），接着把线程 2 的 CONTEXT 信息（即线程 Context_2 中的寄存器信息）放入 CPU 中，继续运行线程 2。

根据上面对硬件断点、线程切换与 CPU 的关系以及 S.E.H 和 V.E.H 相关知识的介绍，我们可以大致按照下面的步骤来设计一个基于硬件断点的 Hook。

（1）创建一个新线程 WorkThread 来负责设置或取消硬件断点。

（2）枚举目标进程中除了 WorkThread 线程以外的线程，暂时称其为目标线程集合 DstThread。

（3）对线程集合 DstThread 中的每个或某个线程执行设置或取消硬件断点操作。

WorkThread 的伪代码示例如下。

```
DWORD WorkThread(DWORD dwSetOrClear)
{
   If(dwSetOrClear == TRUE)
{
        // 设置硬件断点
        do
         {
            // 获取一个线程快照
            hThreadSnap = CreateToolhelp32Snapshot( … );
            // 枚举线程，设置硬件断点
            Thread32First( hThreadSnap, &te);
if( te. th32ThreadID != WorkThreadID)
                // 打开目标线程，获取句柄
                hThread = OpenThread(…, te. th32ThreadID);
                CONTEXT thread_context = {CONTEXT_DEBUG_REGISTERS};
                // 设置 Dr0 为 Hook 后的目标函数地址
                thread_context.Dr0 = FUNC_HOOK_JMP_ADDR;
                thread_context.Dr7 = (1 << 0);
// 设置目标线程的 CONTEXT
SetThreadContext(hMainThread, &thread_context);
```

```
    }
    while{是否可以获取下一个线程}
    }
    }
```

下面让我们看看硬件断点配合 S.E.H/V.E.H 机制后的 Hook 方案。

6.5.4 S.E.H Hook

为了演示 S.E.H Hook 和 V.E.H Hook，下面笔者结合示例程序 MemAccessHook.exe 来进行描述，其界面如图 6-17 所示。

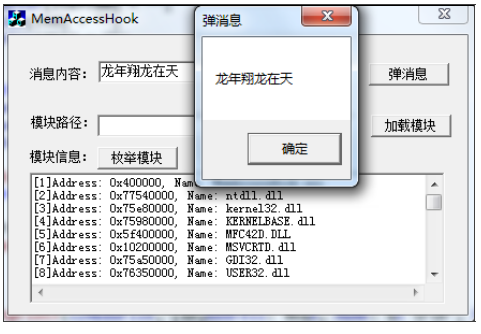


图 6-17 MemAccessHook.exe 主界面

MemAccessHook.exe 程序主要有以下 3 个功能。

- 弹消息：主要负责读取消息内容编辑框中的内容，然后调用 MessageBox 函数来显示。
- 加载模块：调用 LoadLibrary 函数加载模块路径编辑框中提供的模块。
- 枚举模块：枚举当前进程加载的所有模块信息。

后续开发的 VEHHook.dll 和 SEHHook.dll 都会通过“加载模块”按钮加载到 MemAccessHook 程序中，并对 MessageBox 函数设置硬件断点，然后替换消息内容编辑框中的显示内容，达到 Hook 的目的。

S.E.H Hook 中的一个关键函数是 SetUnhandledExceptionFilter，该函数负责取代进程中所有线程的顶层异常处理器，其原型示例如下。

```
LPTOP_LEVEL_EXCEPTION_FILTER WINAPI SetUnhandledExceptionFilter(
  __in  LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter
);
```

参数 lpTopLevelExceptionFilter 指向一个位于顶层的异常过滤函数。当进程处于未调试状态，且进程中有异常发生而未处理时，lpTopLevelExceptionFilter 会接管异常，并对异常进行处理。

S.E.H Hook 的核心代码如下。

```
// 为启动时间最早的线程设置硬件断点
void SEHHook(void)
{
    // 获取线程快照
    HANDLE hTool32 = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
    if(hTool32 != INVALID_HANDLE_VALUE)
    {
        THREADENTRY32 thread_entry32;
        thread_entry32.dwSize = sizeof(THREADENTRY32);

        FILETIME exit_time, kernel_time, user_time;
        FILETIME creation_time;
        FILETIME prev_creation_time;

        prev_creation_time.dwLowDateTime = 0xFFFFFFFF;
        prev_creation_time.dwHighDateTime = INT_MAX;

DWORD    dwThreadEntryOffset    =    FIELD_OFFSET(THREADENTRY32,
th32OwnerProcessID) + sizeof(thread_entry32.th32OwnerProcessID);
        HANDLE hMainThread = NULL;
        if(Thread32First(hTool32, &thread_entry32))
        {

            do {          // 取最早启动的线程作为 Hook 对象
```

```
                if(thread_entry32.dwSize >= dwThreadEntryOffset
                  && thread_entry32.th32OwnerProcessID == GetCurrent
ProcessId()
                  /*&&  thread_entry32.th32ThreadID  !=  GetCurrent
ThreadId()*/)
                {
                            HANDLE hThread = g_lpfnOpenThread(
                            THREAD_SET_CONTEXT   |
                            THREAD_GET_CONTEXT   |
                            THREAD_QUERY_INFORMATION,
                            FALSE, thread_entry32.th32ThreadID);
             GetThreadTimes(hThread,
    &creation_time, &exit_time,
   &kernel_time, &user_time);
                     if(CompareFileTime(&creation_time,
&prev_creation_time) == -1)
                     {
                     // creation_time 小于 prev_creation_time 时候为-1
                     memcpy(&prev_creation_time,&creation_time,sizeof
(FILETIME));
                       if(hMainThread != NULL)
                           CloseHandle(hMainThread);
                       hMainThread = hThread;
                   }
                   else
                     {
                         CloseHandle(hThread);
                     }

                }
                thread_entry32.dwSize = sizeof(THREADENTRY32);
             } while(Thread32Next(hTool32, &thread_entry32));

              // 设置顶层异常过滤器
              (void)SetUnhandledExceptionFilter(ExceptionFilter);
```

```
                CONTEXT thread_context = {CONTEXT_DEBUG_REGISTERS};
                thread_context.Dr0 = func_addr;
                thread_context.Dr7 = (1 << 0);
                // 设置线程 CONTEXT
                SetThreadContext(hMainThread, &thread_context);
                CloseHandle(hMainThread);
            }
            CloseHandle(hTool32);
        }
    }
```

至于 SEHHook() 函数中的 ExceptionFilter() 函数等其他代码的情况，请读者参考本章资源包中的 SEHHook 工程，其代码量较小。当然，对于 SEHHook() 函数中的设置硬件断点部分，可以指定特定线程，或全部线程，或排除自己的线程 —— 这些都可以根据需求来设置。

SEHHook() 函数被 MemAccessHook 加载后单击“弹消息”按钮的效果如图 6-18 所示。可以看到，字符串“龙年翔龙在天”被替换成了“WINSUN”，这说明 S.E.H Hook 成功了。

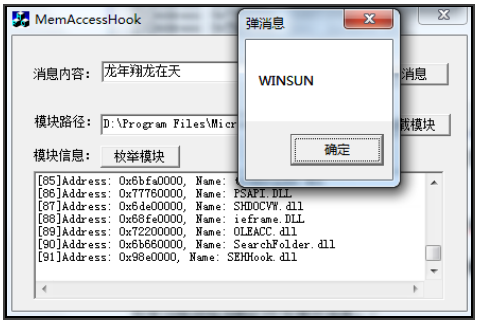


图 6-18　S.E.H Hook MessageBoxA 函数后的运行效果

6.5.5　V.E.H Hook

V.E.H Hook 的过程与 S.E.H Hook 基本一致，只不过设置异常处理器的函数不一样。V.E.H Hook 使用的是 AddVectoredExceptionHandler 函数，这个函数在第 6.5.2 节

中已经做了介绍，具体代码参见本章资源包中的 VEHHook 工程。

高级 Hook 的内容涉及硬件断点、S.E.H 机制和 V.E.H 机制，这些技术结合使用可以产生新的 Hook 方式。虽然新的 Hook 方式比较隐蔽，但凡事都有利弊——由于硬件断点有限制，所以只能 Hook 4 个不同的地址（即 DR0 ~ DR3），而且新的 Hook 方式并非不可检测，这就是安全中的“道高一尺，魔高一丈”。

6.5.6 检测 V.E.H Hook

有些时候，我们会在 ring3 级遇到很诡异的劫持现象，却苦于找不到被修改的代码。这个时候，有必要检测一下是否存在 V.E.H Hook（当然，检测线程的 CONTEXT 是否有硬件断点是更直接的方法）。但是，对于 V.E.H，微软并没有提供专门的 API 来读取，只提供了增加和删除 V.E.H 的接口。所以，在这种情况下，就需要具有创造性的方案了。

本节提供的方案通过注册假 V.E.H 节点的方法来获取 V.E.H 的链节点指针。这种思路其实在 NDIS Hook 中也出现过。下面让我们一起看看检测 V.E.H Hook 的示例代码。

```
    // 检测 V.E.H 是否存在
    VOID CheckVEHHook(VOID)
    {
       CHAR szOut[MAX_PATH] = {0};
       __try
       {
          PVOID pExceptionHandle = NULL;
          PVECTORED_EXCEPTION_NODE pCurrentNode = NULL;
          PVECTORED_EXCEPTION_NODE pNextNode = NULL,pDelNode = NULL;
          // 1、注册 V.E.H 获取 V.E.H 的节点，从尾部插入
          pCurrentNode     =     (PVECTORED_EXCEPTION_NODE)AddVectored
ExceptionHandler(0, (PVECTORED_EXCEPTION_HANDLER)PageHookHandler);
          if ( pCurrentNode == NULL )
          {
             OutputDbgInfo(("[-] AddVectoredExceptionHandler fail !"));
```

```
            return;
        }

        // 2、CurrentNode 是插入尾部后面的新注册的节点
        pNextNode = (PVECTORED_EXCEPTION_NODE)pCurrentNode->ListEntry.
Flink;

        // 3、越过头节点
        pNextNode = (PVECTORED_EXCEPTION_NODE)pNextNode->ListEntry.
Flink;

        // 4、枚举其他 V.E.H
        for (;
            pNextNode != pCurrentNode;
             )
             {
                 pExceptionHandle = DecodePointer(pNextNode->pfnHandler);
                 if ( pExceptionHandle )
                 {

                     pDelNode = pNextNode;
   pNextNode = (PVECTORED_EXCEPTION_NODE)pNextNode->ListEntry.Flink;
                     if(RemoveVectoredExceptionHandler((PVOID)
pDelNode))
                     {
      sprintf(szOut," 注 册 的 异 常 地 址 = 0x%.8x 被 成 功 删 除 ",
pExceptionHandle);
                         AddMsgToList(szOut);
                     }
                     else
                     {
      sprintf(szOut," 注 册 的 异 常 地 址 = 0x%.8x 删 除 失 败 ",
pExceptionHandle);
                         AddMsgToList(szOut);
```

```
                    }
                    OutputDbgInfo((szOut));

                }
            }

// 删除已经注册的 V.E.H
// 可以删除自己的假节点
        //RemoveVectoredExceptionHandler((PVOID)pCurrentNode);

    }
    __except(EXCEPTION_CONTINUE_EXECUTION)
    {
        OutputDbgInfo(("[-] CheckVEHHook exception!"));
    }
}
```

6.6 本章小结

本章先介绍什么是 Hook，接着给出 Hook 技术大致的分类（包括 Inline Hook 和非 Inline Hook），然后讲解 Hook 技术在 IAT 和虚表等中的强大作用，以及 Detour 技术和基于 Detour 的 Hook 引擎，最后进一步提出了基于 S.E.H 和 V.E.H 的高级 Hook 技术。

本章内容可谓 ring3 级的 Hook 技术大全，希望读者能掌握其中精髓，融会贯通，提出更隐蔽的 Hook 和 Hook 检测技术。

第 7 章　应用层防护

本章主要关注外挂应采取的自我保护方法。当然，PE 程序的保护方法很多，包括加密、加壳、反调试、混淆等。在本章中，笔者会站在一个反外挂的逆向分析人员的角度来介绍一些更实际和更有效的保护方案。

经过前面 6 章的学习，相信读者已经对游戏安全技术有了一定的认识，在外挂攻防对抗中也知道如何能更好地隐藏模块、交互、Hook、Call 等。不过，只有经过本章的补充，才可以完整地发行我们的软件。

下面让我们从宏观的角度来看看一款外挂的常见功能，如图 7-1 所示。

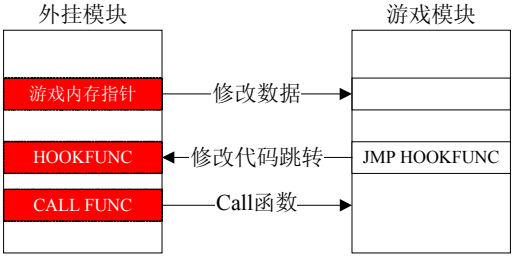


图 7-1　外挂的功能

外挂的常见功能包括 Hook（修改代码跳转）、修改数据（修改血量等）和 Call 函数，每个功能都可能暴露自己的模块地址和核心逻辑的位置。Hook 由于会改变游戏模块代码段内存的完整性，所以很容易被发现，而且这个跳转的目标地址有可能

暴露外挂模块的地址和核心逻辑的地址。Call 函数容易被堆栈回溯检测发现，同样也会暴露外挂模块的地址。而对修改数据功能，由于外挂模块必定会保留游戏模块中某个数据结构的全局指针，所以也很容易暴露外挂模块的地址和核心逻辑的地址。

由于外挂的这些功能有暴露自己的风险，所以下面我们会从静态保护和动态保护两个方面来阐述如何避免这些风险。对于 Call 函数的反堆栈回溯策略，已经在第 5 章中给出了伪栈帧的绕过方案。

7.1 静态保护

本节并不是要告诉读者应该如何加壳、用什么壳，而是介绍去静态特征，包括但不限于去字符串、去全局指针等。关于壳和虚拟保护的软件及方案网上有很多，读者可以视情况采用，这里就不阐述了。

在本节中，笔者特别开发了两个演示程序来说明去静态特征的过程。

SelfProtectForStatic.exe 程序的运行界面如图 7–2 所示。

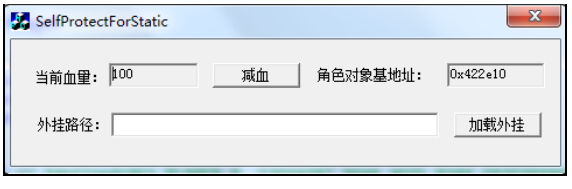


图 7-2 SelfProtectForStaitc.exe 的运行界面

SelfProtectForStaitc.exe 运行之后，会读取同一目录下的配置文件 SPFS.ini（其中包括角色血量的初始值），然后创建角色（Player）对象，该对象定义如下。

```
class Player
{
public:
    Player();
    virtual ~Player();
    int m_iHP;    // 角色血量
};
```

用 SPFS.ini 中的初始血量来初始化角色对象并显示在界面上。单击“减血”按钮之后，会将角色对象中 m_iHP 的值减 1，并更新到界面上。图 7–2 中的“角色对象基地址”是角色对象在内存中的地址，单击“加载外挂”按钮可以加载修改血量的外挂模块。

ChangePlayerHP.dll 程序主要负责动态地将角色对象中 m_iHP 的值增加 1 000。单击“加载外挂”按钮加载 ChangePalyerHP.dll 后，当前的血量值如图 7–3 所示。

图 7-3　加载增加血量的外挂

下面我们来看看 ChangePlayerHP.dll 程序的示例代码。

```
PDWORD g_pdwPlayer = (PDWORD)0x422e10;

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                       )
{
    char szBuf[MAX_PATH] = {0};
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:

        // 增加血量
        *(g_pdwPlayer+1) = *(g_pdwPlayer+1) + 1000;
        sprintf(szBuf, "成功加血 : %d ", 1000);
        OutputDebugString(szBuf);
        break;
    }
```

```
        return TRUE;
    }
```

不知道从逆向的角度来看，上面的代码会暴露给外挂分析人员多少信息。从静态特征来看，至少包括以下 3 处。

- 角色基地址为 0x422e10。
- 增加的血量值为 1 000。
- 字符串“成功加血 ：%d”。

从这 3 处中的任何一处出发，都能很快定位到修改 SelfProtectForStatic.exe 程序的角色对象的 m_iHP 处。

我们可以使用 IDA 的搜索功能来搜索特征，然后通过 IDA 的交叉引用功能，就能很快定位外挂的核心逻辑。

下面让我们看看通过静态特征如何具体定位修改血量的逻辑。

（1）搜索“422E10”，如图 7–4 所示。通过【Alt】+【B】组合键可以打开 IDA 的 Binary Search 功能，输入“422E10”之后可以找出内存中存放该值的地址，如图 7–5 所示。双击相应地址，就可以定位对“422E10”存在交叉引用的地方，如图 7–6 所示。单击其中的交叉引用，就可以判断这里是否是我们需要选择的关键修改逻辑处，如图 7–7 所示。

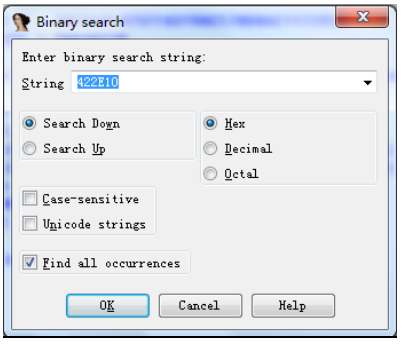


图 7-4 搜索角色基地址

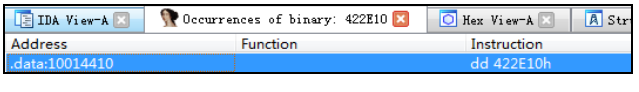


图 7-5 定位角色指针存放地址

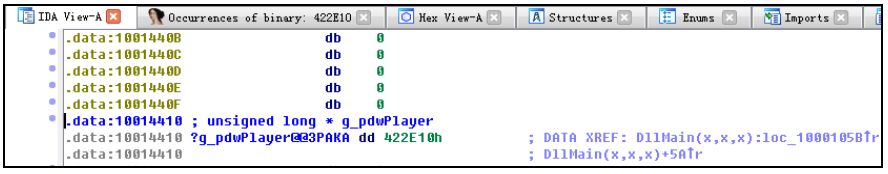


图 7-6　定位交叉引用角色指针的位置

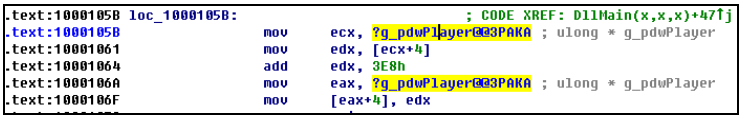


图 7-7　定位修改血量的位置

（2）按下【Alt】+【B】组合键搜索“0x3E8(1000)”。因为 IDA 中的值在默认情况下是十六进制，所以可以先把十进制数 1 000 转换为十六进制数 3E8，然后通过 Binary Search 功能搜索“3E8”，如图 7–8 所示。根据 add 指令和包含“3E8”的两条信息，我们就可以双击条目查看相应的逻辑了，如图 7–9 所示。

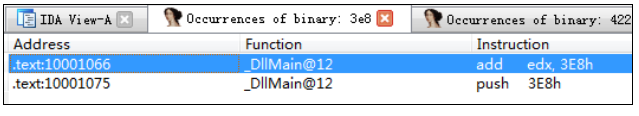


图 7-8　定位引用 1 000 的代码

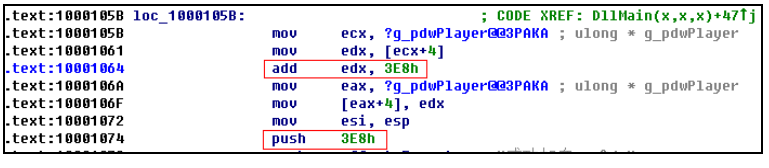


图 7-9　定位修改血量的位置

（3）定位关键字符串。按下【Shift】+【F12】组合键，可以扫描出程序中用到字符串，如图 7–10 所示。很明显，双击第一个条目就能定位交叉引用字符串的位置，如图 7–11 所示，通过此交叉引用可以快速定位如图 7–12 所示的关键修改逻辑。

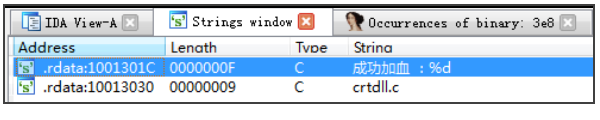


图 7-10　定位字符串

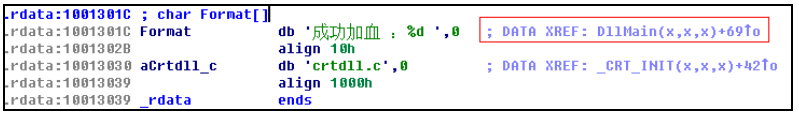


图 7-11 交叉引用关键字符串的位置

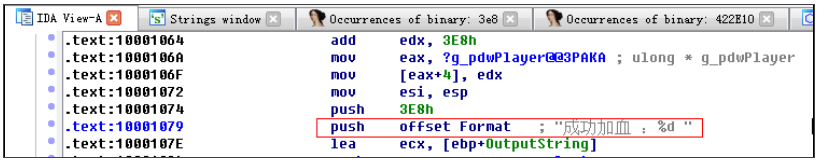


图 7-12 通过关键字符串定位修改逻辑

通过上面的分析可知，要想保护自己的程序不被分析，哪怕一个小细节也是不能忽略的。对于去字符串，比较容易解决——反正是自己写的程序，只要不留下任何打印信息就行。但是，对于角色基地址这种常量，我们应该怎么做呢？其实，去常量的方法也比较多，最简单和最常见的就是拆分成多个值运算以返回运算结果，而不是把常量值保存在数据段中。

例如，对于 0x422E10，我们可以这样做。

```
DWORD dwValue = 0x845C20;
PDWORD pdwPlayer = (PDWORD)dwValue /2;
```

当然，去静态特征的方法很多，而且静态特征不止这些，本节讲的都是最基础的。但是，笔者分析过的软件中有很多还是会犯这些错误。

7.2 动态保护

在本节中，我们主要关注如何防止模块被 dump，以及绕过 Hook 所带来的完整性校验问题。

7.2.1 反 dump

很多外挂作者可能未曾考虑反 dump。但事实上，当动态调试对于外挂分析人员来说效果并不明显的时候，就需要 dump 外挂模块，放入 IDA 进行静态辅助分析。

在脱壳和反脱壳领域，反 dump 技术已经比较成熟了，很多经典的文章中都有论述，其中包括本章资源包中的两篇文章——《现代 dump 技术及保护措施》和 *Win32 Portable Executable Packing Uncovered*。

在 dump 模块的方法中，一般会调用 ReadProcessMemory 函数来读取模块的内容。这个函数除了需要知道模块的起始地址，还需要获取模块的大小，而模块的大小一般是通过 PE 结构的可选头中的 ImageOfSize 来获取的。所以，只要对 ImageOfSize 进行恶意修改，例如改得特别大或特别小，dump 的时候肯定会出问题。显然，一劳永逸的方法是直接抹掉模块的 PE 头，但是，这种方法对直接指定地址和大小的 dump 来说就不起作用了。

简单而有效地反 dump，有一个方法比较好 —— 修改页面的访问标志为 PAGE_NO_ACCESS。下面就让我们一起通过一个示例程序来了解它的实现方法。

这个示例程序涉及如下两个模块。

- SelfProtectForDynamic.exe：负责枚举、加载和 dump 模块。
- PageNoAccessDll.dll：负责保护数据节不被 dump。

下面演示反 dump 的过程。

（1）如图 7-13 所示，已经加载了 PageNoAccessDll.dll 模块。

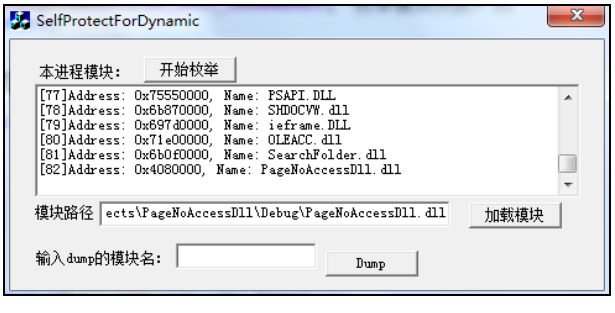


图 7-13　SelfProtectForDynamic 主界面

（2）如图 7-14 所示，已经成功 dump kernel32.dll 模块。

（3）如图 7-15 所示，dump PageNoAccessDll.dll 模块失败，原因是内存不可读。

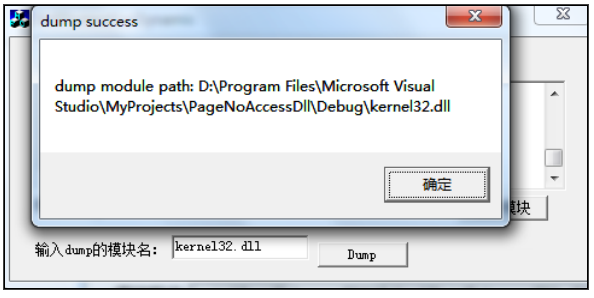


图 7-14 dump kernel32.dll

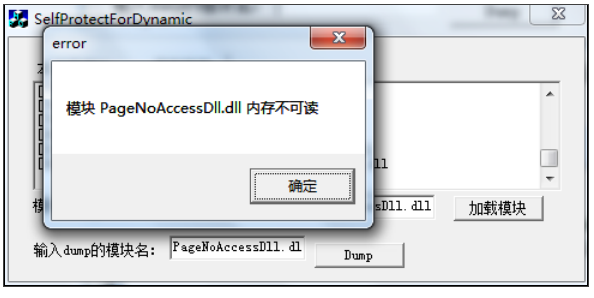


图 7-15 dump PageNoAccessDll.dll

为了让读者了解上述过程的原理，下面让我们一起看看 PageNoAccessDll 模块实现反 dump 功能的代码。

```
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                       )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        ChangeDataSectionPageProtectAttr(PAGE_NOACCESS);
        break;
    }
    return TRUE;
}
```

在模块被加载的时候，DllMain 调用 ChangeDataSectionPageProtectAttr() 函数来改变数据节所在页面的保护属性。ChangeDataSectionPageProtectAttr() 函数的示例如下。

```
    DWORD ChangeDataSectionPageProtectAttr(DWORD dwProtect)
    {
        DWORD dwOldProtect;
        MEMORY_BASIC_INFORMATION mbi = {0};
        __try
        {

            VirtualQuery(    ChangeDataSectionPageProtectAttr,    &mbi,
sizeof(mbi) );
        VirtualProtect(  (LPVOID)((PBYTE)mbi.BaseAddress+mbi.RegionSiz
e), 1024, dwProtect, &dwOldProtect  );
        }
        __except(EXCEPTION_CONTINUE_EXECUTION)
        {
            OutputDebugString("ChangeDataSectionPageProtectAttr
exception \r\n!");
        }
        return dwOldProtect;
    }
```

上面的函数先通过 VirtualQuery() 函数查询代码节的起始地址和大小等内存信息，再通过 VirtualProtect() 函数将代码节后面的节所在页面的保护属性更改为 PAGE_NOACCESS。通常情况下，代码节之后是数据，对于 dump 整个模块来说，由于数据节不可访问，因此阻止了内存读取操作，也就阻止了 dump 模块的行为（如图 7–15 所示）。

在 SelfProtectForDynamic 模块中，dump 操作的核心步骤如下。

（1）枚举目标模块。

（2）分配内存。

（3）读取目标模块的内容，将其存入上一步中分配的内存。

（4）创建文件并把目标模块内容写入文件。

SelfProtectForDynamic 模块的具体代码参见本章资源包中的 SelfProtectForDynamic 工程。

7.2.2 内存访问异常 Hook

通过对第 6 章的学习，相信读者对 Hook 技术应该比较了解了，特别是第 6.5 节提到的基于 V.E.H 或 S.E.H 的硬件断点 Hook。读者可能会问，既然已经有这么多 Hook 技术了，为什么还要设计内存访问异常 Hook 的机制呢？原因很简单：第一，修改代码方式的 Hook 很难绕过代码检验；第二，硬件断点触发的异常可能被通过 GetThreadContext 函数获取硬件断点的设置检测出来，而且硬件断点只有 4 个。

在本节中，我们将通过设置代码所在页为 PAGE_NOACCESS 来触发异常，然后通过之前注册的 V.E.H 或 S.E.H 来接管异常的方式实现 Hook。另外，本节也将通过 MemAccessHook.exe 程序和 MemHook.dll 来演示内存访问异常 Hook。

MemAccessHook.exe 程序在第 6 章中已经给出运行情况和代码，下面主要介绍一下 MemHook.dll 的实现，其运行效果如图 7-16 所示。

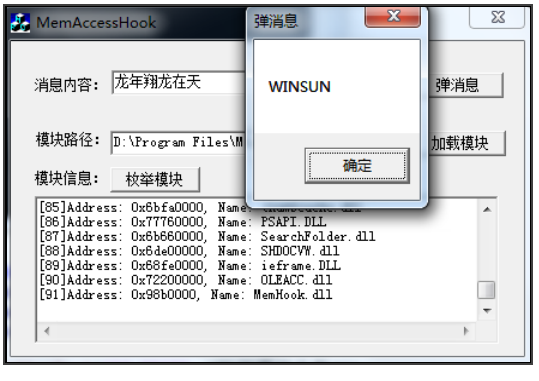


图 7-16 内存访问异常 Hook

MemAccessHook.exe 程序加载 MemHook.dll 后，单击“弹消息”按钮，因为 MessageBoxA 被 Hook 后修改了显示内容，所以这里显示的是“WINSUN”，而不是“龙年翔龙在天”。

MemHook.dll 的主要步骤如下。

（1）获取 MessageBoxA，示例如下。

```
    FUNC_ADDR = (DWORD)GetProcAddress(GetModuleHandle("user32.dll"),
    "MessageBoxA");
    FUNC_ADDR_OFFSET = FUNC_ADDR + 2;  //2 是 mov edi, edi 指令的长度
```

（2）调用 MemHook 函数，然后注册 V.E.H，修改函数所在页面属性为“PAGE_NOACCESS”，示例如下。

```
    DWORD MemPageHook(DWORD dwNewFuncAddr)
    {
        // 注册 V.E.H
        g_AddVectorExceptionHandler  =
    (ADDVECTOREEXCEPTIONHANDLER)GetProcAddress(GetModuleHandle("kern
el32.
    dll"), "AddVectoredExceptionHandler");
        g_AddVectorExceptionHandler(1, ExceptionFilter);

        // 修改目的地址所在页面的保护属性
        DWORD dwOldProtect;
        VirtualProtect((PVOID)dwNewFuncAddr,     4,     PAGE_NOACCESS,
&dwOldProtect);
        return dwOldProtect;
    }
```

（3）ExceptionFilter 函数接管异常，设置 Hook，恢复原始代码所在页面的访问属性，示例如下。

```
    LONG WINAPI ExceptionFilter(PEXCEPTION_POINTERS ExceptionInfo)
    {
        DWORD dwProtect;

    // 过滤其他异常
    if (ExceptionInfo->ExceptionRecord->ExceptionCode != STATUS_ACCESS
_VIOLATION
```

```
    &&ExceptionInfo->ExceptionRecord->ExceptionCode != STATUS_SINGLE_
STEP)
       {
           return EXCEPTION_CONTINUE_SEARCH;
       }

       // 判断单步内存访问异常的地址是否是要 Hook 的函数地址
       if((DWORD)ExceptionInfo->ExceptionRecord->ExceptionAddress  ==
FUNC_ADDR) {
              PCONTEXT debug_context = ExceptionInfo->ContextRecord;

              // 设置由 EIP 实现 Hook
              printf("Breakpoint hit!\n");
              PrintParameters (debug_context);
              ChangeText (debug_context);
              debug_context->Eip = (DWORD)& ReturnOrigianlFunc;

               // 恢复原始代码地址的页面访问属性
               VirtualProtect((PVOID)FUNC_ADDR, 4, PAGE_OLD_PROTECT,
    &dwProtect);
         }
          return EXCEPTION_CONTINUE_EXECUTION;
    }
```

更详细的代码参见本章资源包中的 MemHook 工程代码，这里就不赘述了。

内存访问异常 Hook 比之前提到的所有 Hook 技术更加隐蔽，但可能存在因某个代码页被频繁访问而影响原始程序性能的问题。

7.3 本章小结

虽然本章提到的技术从实现上来说都不难，但它们往往是很多软件作者忽略的地方。只要把这些都考虑到，从某种程度上来说，我们已经能够编写一款自我保护能力不错的软件了。

第 3 篇

游戏保护方案探索篇

第 8 章　探索游戏保护方案

第 8 章　探索游戏保护方案

游戏安全领域的保护系统比较多，各个游戏公司会根据自己的需求来选择采用第三方开发的系统或自己研发一套系统，这里就不一一介绍了。

在本章中，笔者会以某款游戏的保护方案为例，在详细分析其保护方案的基础上，提出如何分析未知游戏的方案。在整个探究游戏保护方案的过程中，主要使用两款分析工具—— GameSpider 和 Kernel Detective，本章也会详细介绍它们的用法。

8.1　分析工具介绍

古语有云：工欲善其事，必先利其器。对一名逆向分析者来说，拥有一款方便、快捷的分析工具，无疑是如虎添翼的。

本节将带领读者快速了解的两款强有力的分析工具—— GameSpider 和 Kernel Detective。

8.1.1　GameSpider

GameSpider（简称 GS）是一款 Windows 平台下的外挂和游戏辅助分析工具。它是一个 DLL 模块，需要注入一个 GUI 程序中，界面如图 8-1 所示。

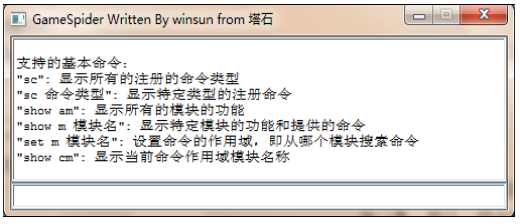


图 8-1　GameSpider 的界面

因为目前无论是外挂还是游戏，都加强了反调试策略，分析的时候往往会阻塞调试器，从而妨碍调试，又因为在 Windows 平台上注入一个模块相对比较简单，所以，综合起来看，编写一个寄生模块，使其与被分析程序在同一个进程空间，似乎是个不错的想法——GS 就这样诞生了。

GS 目前支持 13 种类型的命令，如图 8-2 所示。本章的资源包中提供了 GS 程序。

下面介绍一下分析中常用的 5 类命令。

1. Mem

Mem 类型的命令主要是指与内存相关的一些操作，目前支持 6 种命令，如图 8-3 所示。eph 命令用于显示进程堆信息；pms 命令用于以字符串形式打内存补丁；pmh 命令用于以十六进制形式打内存补丁；dms 命令用于显示内存节信息；das 命令用于显示某地址所处的节；dm 命令用于显示某地址处的内存。

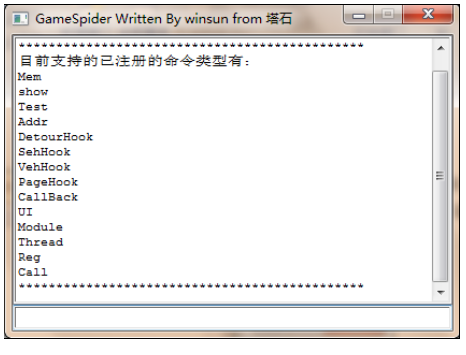


图 8-2　GS 命令的类型

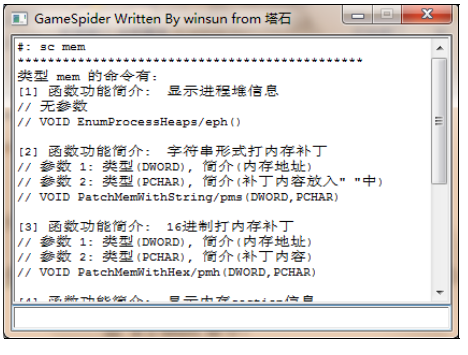


图 8-3　Mem 命令

2. Addr

Addr 命令主要是指与内存地址操作相关的命令，目前支持 4 种命令，如图 8-4 所示。da 命令是反汇编默认长度指令；dal 命令是反汇编指定长度指令；prf 命令用于输出注册函数地址；ga 命令用于获取导出函数地址。

因为 GS 的设计基于动态注册机制，即可以动态注册函数信息供用户使用，所以在注册函数时，只需要输入模块名字、函数名字、参数类型、参数数目和函数地址。

3. DetourHook

DetourHook 命令主要提供 Detour Hook 功能，包括动态 Hook/Unhook 地址、打印堆栈回溯信息等，目前支持 6 种命令，如图 8-5 所示。udh 命令用于卸载 Detour Hook；dhc 命令用于禁止劫持之后调用回调函数；ehc 命令用于劫持之后调用回调函数；edh 命令用于枚举 Detour Hook 信息；hac 命令用于实施 Detour Hook 并打印堆栈回溯信息；ha 命令用于实施简单的 Detour Hook，但不打印堆栈回溯。

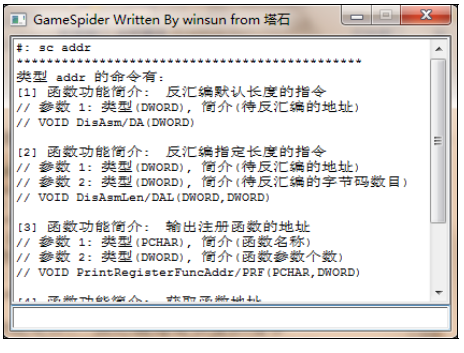


图 8-4 Addr 命令

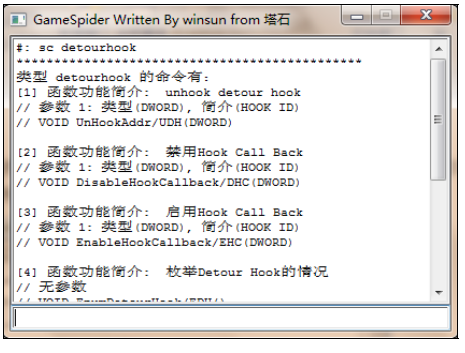


图 8-5 DetourHook 命令

4. Module

Module 类型的命令主要是指与模块处理相关的操作。这个类型的命令比较多，有 16 种，如图 8-6 所示是部分命令。

5. Thread

Thread 命令支持悬挂或恢复线程、枚举线程等功能，目前支持 5 种命令，如图 8-7 所示。

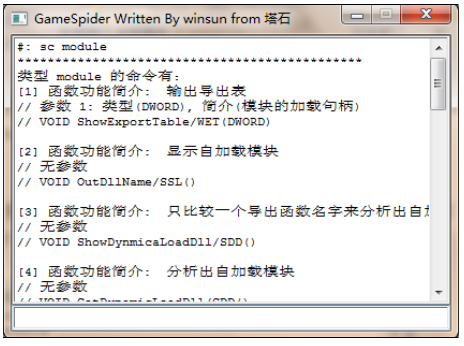


图 8-6　Module 命令

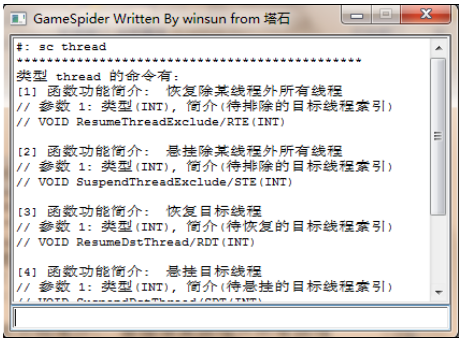


图 8-7　Thread 命令

在 Thread 命令中，gtl 命令不仅能看到线程的起始地址、状态等基本信息，还能看到线程属于哪个模块，以及线程是否被设置了硬件断点。如果硬件断点字段是 DR0 ~ DR3，那么肯定设置了硬件断点。

下面给出一个显示线程信息和反汇编特定地址的示例。在 GS 命令输入框中输入“gtl”并按【Enter】键，可以查看线程的相关信息，如图 8–8 所示。其他线程相关函数可能涉及输入参数，其参数和命令之间以及参数和参数之间都用空格分割。

下面再让我们看看反汇编特定地址的情况。da 命令用于反汇编默认长度的指令，执行“da 532d6c”指令显示的结果如图 8–9 所示。

图 8-8　线程信息

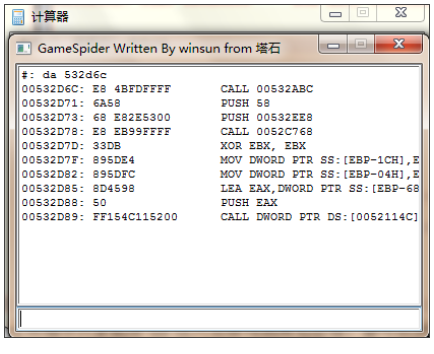


图 8-9　通过 da 命令显示反汇编结果

经过上面对 GS 的简单介绍，相信读者已经知道了如何使用 GS。

GS 的主要组成部分是 GameSpider.dll，任何程序只要能将 GS 注入游戏或其他进程，都可以使用 GS 提供的服务。至于更多的 GS 命令和用法，请读者自己动手尝试。

8.1.2　Kernel Detective

Kernel Detective 是一款用于检测、分析和修改 Windows NT 内核的免费工具，读者可以到 http://www.at4re.com/news.php?item.14.2 下载它。Kernel Detective 的操作界面如图 8-10 所示。

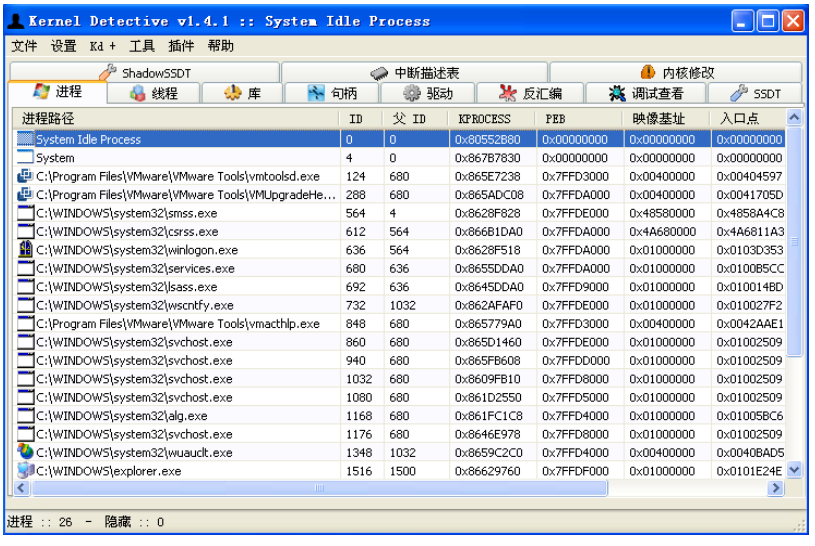


图 8-10　Kernel Detective 的操作界面

Kernel Detective 是一款非常强大的静态逆向分析工具，其具体用法在这里就不多说了，请读者自己尝试使用。

在认识了上述两款分析工具之后，下面就让我们试着去分析和了解游戏的保护方案。

8.2　定位保护模块

定位游戏的保护模块是分析游戏的前提，因为只有摸清谁是保护模块，并掌握其保护思路，才能有的放矢地突破这些保护。

本节会按 ring0 级到 ring3 级的顺序来阐述如何定位保护模块。

8.2.1　定位 ring0 保护模块

一款游戏本身一般是不会引入驱动作为游戏功能的一部分的，而往往是因为反外挂安全模块需要取得底层的控制权才引入的，所以，定位 ring0 级的保护模块（驱动）就显得尤为重要。

下面让我们看看如何采用差异分析思想，通过 Kernel Detective（以下简称 KD）来为定位 ring0 保护模块。

KD 有一个显示系统当前加载的驱动列表的功能，如图 8-11 所示。

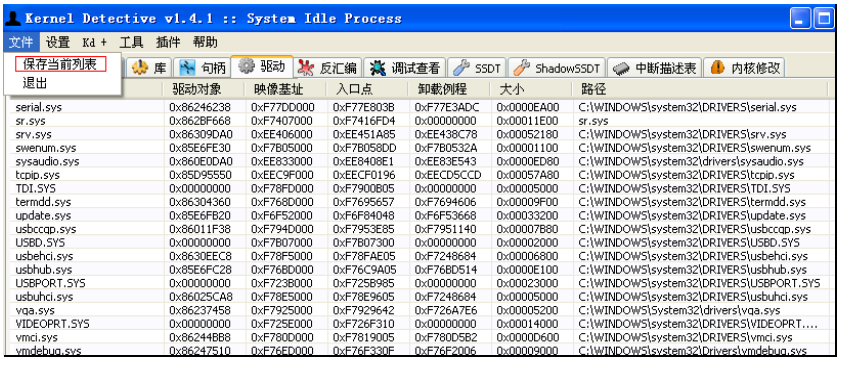


图 8-11　KD 驱动列表

游戏加载前和加载后，KD 分别保存驱动加载列表，这样就可以获得两份文件列表，即 unload.txt 和 load.txt。

现在，我们用 Beyond Compare（以下简称 BC）来比较 unload.txt 和 load.txt。BC 的操作界面如图 8-12 所示。

在 BC 中，暂时被抹掉的文件名称和引用字段叫做 X.SYS。可以看到，游戏加载之后，load.txt 中多了一个 X.SYS 驱动，所以，游戏加载了 X.SYS 驱动，而这个驱动正是我们要寻找的 ring0 保护模块。

8.2.2　定位 ring3 保护模块

定位 ring3 保护模块，大致有以下两种思路。

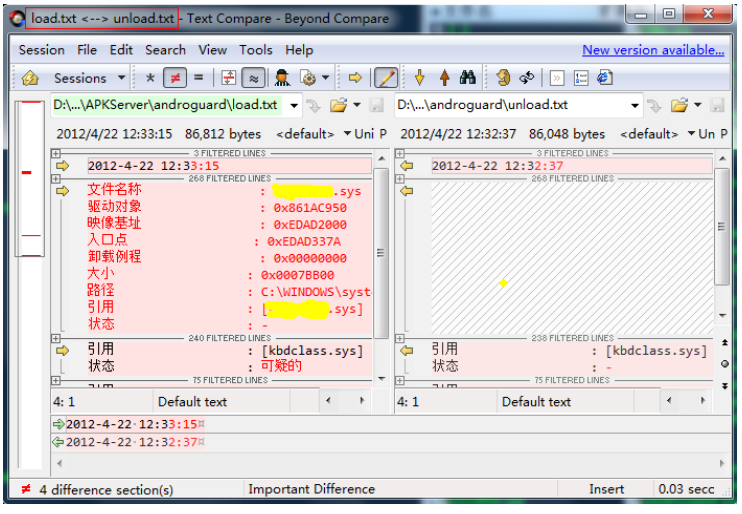


图 8-12　用 BC 比较 load.txt 和 unload.txt

1. 回溯法

我们可以在 ring3 和驱动模块通信的接口处 Hook，然后进行调用回溯，这样做可以回溯到 ring3 保护模块。通常情况下，ring3 级的程序可以调用 CreateFile 打开驱动的设备对象，进而通过 DeviceIoControl 与设备对象通信。

对 DeviceIoControl 接口设置 Detour Hook 并进行堆栈调用回溯的结果如图 8-13 所示。可以看到，3 个层次上回溯的结果都是 DSOUND.dll，所以，从 DeviceIoControl 回溯的方法在这款游戏中不起作用，估计它采用了其他方式与驱动进行通信，或者做了其他保护，而且 Inline Hook 被游戏保护系统发现了（事实上，这款游戏自加载了很多关键模块，后面会提到）。

在这款游戏中，虽然回溯法在 DeviceIoControl 这个点上不起作用，但它仍然是分析中不可丢弃的方法。

2. 差异法

由于保护模块通常会 Hook 操作系统的原生 DLL 接口来进行保护，所以可以通过比较原生 DLL 文件和加载到内存中的原生 DLL 的方法来定位 ring3 保护模块。

在分析的过程中，为了防止被 ring3 保护模块发现，可以暂时把除了当前线程以外的其他线程暂停，如图 8-14 所示。

```
[STACK] 第 1  层 返回地址 0x73e7228a ,属于模块 DSOUND.dll, 参数如下:
        参数1: 0x19ac
        参数2: 0x2f0003
        参数3: 0x3e13fe54
        参数4: 0x18
        参数5: 0x3e13fe90
        参数6: 0x10
        参数7: 0x3e13fe44
[STACK] 第 2  层 返回地址 0x73e72cdc ,属于模块 DSOUND.dll, 参数如下:
        参数1: 0x19ac
        参数2: 0x2f0003
        参数3: 0x3e13fe54
        参数4: 0x18
        参数5: 0x3e13fe90
        参数6: 0x10
        参数7: 0xffffffff
[STACK] 第 3  层 返回地址 0x73e72c65 ,属于模块 DSOUND.dll, 参数如下:
        参数1: 0x19ac
        参数2: 0x73e72c88
        参数3: 0x5
        参数4: 0x3e13fe90
        参数5: 0x10
        参数6: 0x0
        参数7: 0x3427648
```

图 8-13　DeviceIoControl 堆栈调用回溯

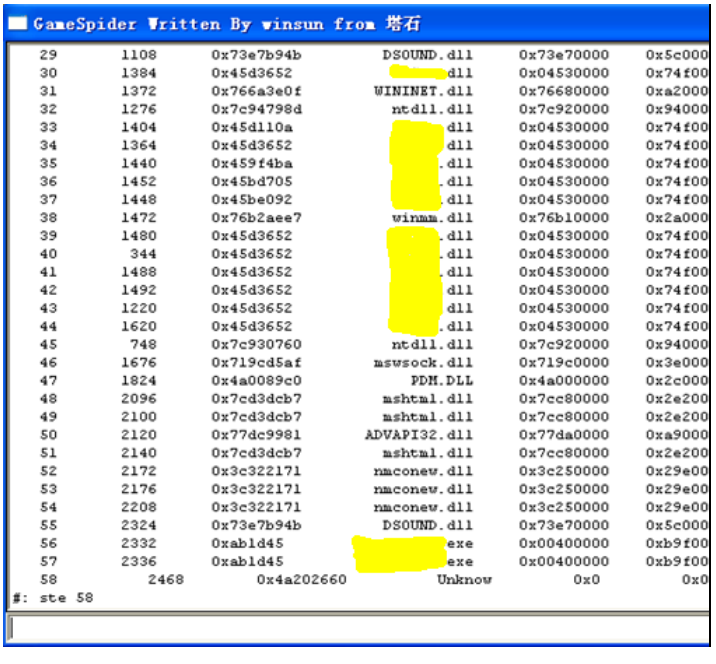


图 8-14　悬挂除当前线程外的其他线程

可以看出，除当前线程外，该游戏有 58 个线程，可以通过 SuspendThread() 函数悬挂这些线程以便后续的分析（GS 的命令就是 ste 序号）。

下面，我们再对 3 个常用的原生 DLL—— ntdll.dll、kernel32.dll 和 user32.dll 进行文件和内存的比较，如图 8-15 所示。

图 8-15　差异分析原生 DLL 的变化

可以看出，ntdll.dll 模块中有 3 处地址发生了变化，分别是 0x7c921230、0x7c92DEB6 和 0x7c97077B。下面让我们看看这 3 处地址目前的指令是什么，如图 8-16 所示。

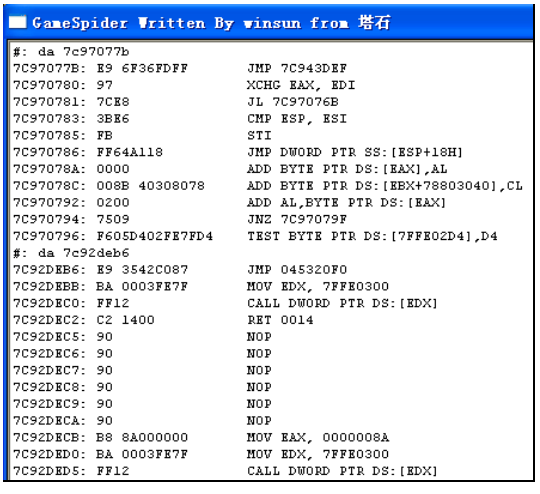


图 8-16　两处地址 jmp 指令

可以看出，ntdll.dll 模块中有一处 jmp 指令跳入地址 0x45320F0。那么，它到底属于哪个模块呢？如图 8-17 所示，从地址区间来看，是包含 0x45320F0 的。所以，在 ntdll.dll 模块中，地址 0x7c92deb6 会跳入 X.dll 模块。

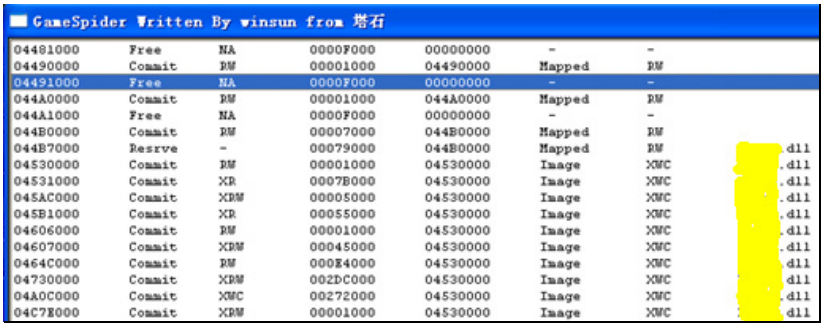


图 8-17　模块节信息

到目前为止，我们基本可以判断——ring3 级的游戏保护模块是 X.dll。

为了能更准确地判断 X.dll 是否确实是保护模块，下面让我们看看 ntdll.dll 模块中 3 处发生变化的地址在原生 ntdll.dll 模块中的作用。

使用 IDA 对 ntdll.dll 模块进行分析，然后定位地址 0x7c921230、0x7c92DEB6 和 0x7c97077B。如图 8-18 所示，原来此处是 DbgBreakPoint() 函数。这个函数是供调试器下软件断点的，而在游戏中却被改成了 ret 指令，这样做能起到防止下软件断点的作用。

```
text:7C921230
text:7C921230                   public DbgBreakPoint
text:7C921230 DbgBreakPoint     proc near               ; CODE XREF: sub_7C93C9E4:loc_7C95AD0A↓p
text:7C921230                                           ; sub_7C941ABC:loc_7C95EDBB↓p ...
text:7C921230                   int     3               ; Trap to Debugger
text:7C921231                   retn
text:7C921231 DbgBreakPoint     endp
text:7C921231
```

图 8-18　地址 0x7c921230 所处函数

如图 8-19 所示，地址 0x7c97077B 是属于 DbgUiRemoteBreakin() 函数的，这个函数的详细介绍可以参考张银奎老师《软件调试》一书的第 10.6.4 节。下面简单说明这个函数的作用。

DbgUiRemoteBreakin() 函数是 ntdll.dll 模块提供的用于在目标进程中创建远线程软件断点的函数，其伪代码如下。

```
DWORD WINAPI DbgUiRemoteBreakin( LPVOID lpParameter)
{
    __try
```

```
{
        if(NtCurrentPeb->BeingDebugged)
            DbgBreakPoint();
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
        Return 1;
}
RtlExitUserThread(0);
}
```

```
.text:7C97077B                public DbgUiRemoteBreakin
.text:7C97077B DbgUiRemoteBreakin:                    ; DATA XREF: .text:off_7C92395C↑o
.text:7C97077B                                        ; DbgUiIssueRemoteBreakin+14↓o
.text:7C97077B                push    8
.text:7C97077D                push    offset dword_7C9707C8
.text:7C970782                call    sub_7C92EDC2
.text:7C970787                mov     eax, large fs:18h
.text:7C97078D                mov     eax, [eax+30h]
.text:7C970790                cmp     byte ptr [eax+2], 0
.text:7C970794                jnz     short loc_7C97079F
.text:7C970796                test    byte ptr ds:7FFE02D4h, 2
.text:7C97079D                jz      short loc_7C9707BF
.text:7C97079F
.text:7C97079F loc_7C97079F:                          ; CODE XREF: .text:7C970794↑j
.text:7C97079F                and     dword ptr [ebp-4], 0
.text:7C9707A3                call    DbgBreakPoint
.text:7C9707A8                jmp     short loc_7C9707BB
```

图 8-19　地址 0x7c97077B 所处函数

当调试器通过 CreateRemoteThread() 函数在目标程序中创建 DbgUiRemoteBreakin 线程的时候，从代码中看，是下了 int 3 软件断点。由于在被调试状态，所以调试器可以捕获这个异常。如果目标程序没有被调试的话，DbgUiRemoteBreakin 中的 S.E.H 显然可以捕获并处理它。所以，游戏保护系统在 DbgUiRemoteBreakin 进行 jmp 操作，很明显是为了防止调试。

我们再来看看保护系统从 DbgUiRemoteBreakin 线程跳到了哪里。

如图 8-20 所示是保护系统从 DbgUiRemoteBreakin 线程跳入执行函数的过程，很明显，LdrShutdownProcess 是一个关闭进程的函数。

在地址 0x7c92DEB6 处，如图 8-21 所示，保护模块对 ZwProtectVirtualMemory 函数进行了 Hook，以防止虚拟内存所在页面的保护属性被修改。

```
.text:7C943DEF                public LdrShutdownProcess
.text:7C943DEF LdrShutdownProcess proc near           ; DATA XREF: .t
.text:7C943DEF
.text:7C943DEF ; FUNCTION CHUNK AT .text:7C943F6B SIZE 0000001C BYTES
.text:7C943DEF ; FUNCTION CHUNK AT .text:7C955E00 SIZE 00000052 BYTES
.text:7C943DEF ; FUNCTION CHUNK AT .text:7C95E061 SIZE 0000004F BYTES
.text:7C943DEF
.text:7C943DEF                push    50h
.text:7C943DF1                push    offset dword_7C943E98
.text:7C943DF6                call    sub_7C92EDC2
.text:7C943DFB                cmp     byte_7C99C030, 0
.text:7C943E02                jnz     loc_7C943F81
.text:7C943E08                xor     esi, esi
.text:7C943E0A                cmp     dword_7C99C12C, esi
.text:7C943E10                jz      short loc_7C943E1F
.text:7C943E12                push    dword_7C99DE70
.text:7C943E18                call    RtlEncodeSystemPointer
.text:7C943E1D                call    eax
```

图 8-20　地址 0x7C943DEF 所处函数

```
.text:7C92DEB6                public ZwProtectVirtualMemory
.text:7C92DEB6 ZwProtectVirtualMemory proc near       ; CODE XREF: sub_7C93D5B7+64↓p
.text:7C92DEB6                                        ; sub_7C93D5B7+101↓p ...
.text:7C92DEB6                mov     eax, 89h        ; NtProtectVirtualMemory
.text:7C92DEBB                mov     edx, 7FFE0300h
.text:7C92DEC0                call    dword ptr [edx]
.text:7C92DEC2                retn    14h
.text:7C92DEC2 ZwProtectVirtualMemory endp
```

图 8-21　地址 0x7c92DEB6 所处函数

8.2.3　定位自加载模块

有些游戏保护系统为了防止逆向调试人员通过对系统原生 DLL（如 kernel32.dll、ntdll.dll 和 ws2_32.dll 等）所导出的关键 API 下断点来定位关键游戏逻辑或保护方案，往往会在 ring3 级对这些原生 DLL 进行自加载处理。

自加载是指模拟 PE 加载器重新加载一份模块到内存中，之后游戏会调用自加载模块的接口，而不会调用原生 DLL 提供的接口，这样，我们在原生 DLL 所提供的接口下断点或 Hook 的话，就不会得到执行控制权。

网上有一篇不错的对内存中加载 DLL 的介绍，参见 http://www.doc88.com/p-807989404789.html。

下面让我们一起来分析一下如何定位自加载模块。

分析自加载模块的思路比较多，而且效果与加保护的方式有关。例如，如果自加载模块没有加壳，那么是否可以直接比较内存中模块代码节的 Hash 呢？在这里，我们可以通过比较模块的导出信息来定位。

我们都知道，DLL 是一种符合 PE 格式的文件，其导出函数的信息保存在导出表 IMAGE_EXPORT_DIRECTORY 中。假设两个功能相同的模块，其导出信息是一致的，这样，我们就可以比较内存中模块的导出信息，如导出函数的个数、所有导出函数的名字是否相同。

如图 8–22 所示是进程中 47623e165a43.tmp 模块的内存块信息。

图 8-22　47623e165a43.tmp 模块的内存块信息

通过解析 47623e165a43.tmp 模块的 IMAGE_EXPORT_DIRECTORY 结构来输出信息，如图 8–23 所示。

图 8-23　47623e165a43.tmp 模块的 IMAGE_EXPORT_DIRECTORY 结构

可以看到，这个模块在 IMAGE_EXPORT_DIRECTORY 中叫做 KERNEL32.dll，它的导出函数和原生 kernel32.dll 模块一致，所以我们有理由相信，47623e165a43.tmp 模块是一个自加载模块。

有如下 3 种方法来定位其他模块的自加载情况。

- 比较所有模块的导出函数列表，看看导出函数的名称是否一致。
- 比较所有模块的第一个导出函数的名称和导出函数的个数是否分别一致。
- 比较所有模块的加载名字和导出表中的名字是否一致。

显然，在上面的 3 种方法中，第一种比较精确，第二种比较粗放，第三种比较简单。

但是在这款游戏中，由于对自加载模块的关键内存块（包含代码和导出表信息）进行了页面保护，不允许读，只允许执行，所以，第一种方法中大幅度的读取动作会被发现，后续还是要用第三种方法来定位，如图 8–24 所示。

图 8-24　自加载模块

8.3　分析保护方案

在这一节里，笔者会对之前定位的模块进行基础分析，目的是深入了解并准确绕过保护方案。

根据前面的分析，上述游戏的保护方案大致分成两个层次，分别是 ring3 保护方案和 ring0 保护方案。

8.3.1　ring3 保护方案

ring3 保护方案的工作流程大致如下。

（1）Inline Hook ntdll.dll 模块中与调试相关的接口。

ring3 保护方案中对 DbgUiRemoteBreakin 和 DbgBreakPoint 的 Hook，使调试器无法创建远线程来捕获异常和下软件断点。所以，为了绕过保护，可以尝试恢复被 Hook 函数的代码。如果保护方案没有检测被 Hook 的代码是否被还原，那么这样做是可行的。

（2）Inline Hook ntdll.dll 模块中页面保护的相关接口。

ring3 保护方案对 ZwProtectVirtualMemory 的 Hook，可能是为了防止修改某些页面的保护属性而设置的。如果考虑自加载模块的页面保护属性，就大致可以知道，这个 Hook 的作用是防止修改自加载模块的保护属性。同理，可以尝试恢复被 Hook 的函数头，看看是否会被检测到。

（3）设置 V.E.H。

因为我们可以通过 V.E.H 来完成不用修改代码的 Hook，所以，在本保护方案中，V.E.H 的作用很可能是通过捕获对自加载模块的异常访问来排除恶意访问。图 8-25

是一个检测和去除 V.E.H 的例子。

图 8-25　V.E.H 检测和清除

当然，仅清除 V.E.H 是很简单的，第 6.5.2 节有相关介绍。不过，重点是我们要想办法恢复自加载模块的页面可读属性，而且要绕过 ZwProtectVirtualMemory 的 Hook。

（4）自加载核心模块。

自加载模块的模块名字不固定且模块自加载，关键内存块只允许执行，不允许读。如图 8-26 所示，becf19844870.tmp 是 kernel32.dll 的自加载模块。

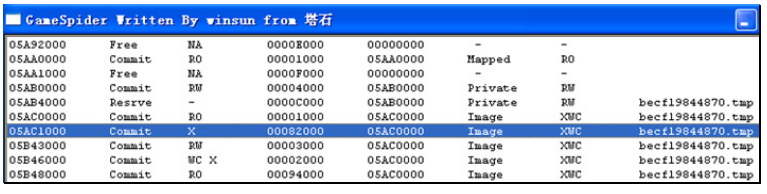


图 8-26　becf19844870.tmp 模块的内存块页面保护属性

证明图 8-26 中的模块是 kernel32.dll，如图 8-27 所示，becf19844870.tmp 模块实际上是 kernel32.dll 模块。

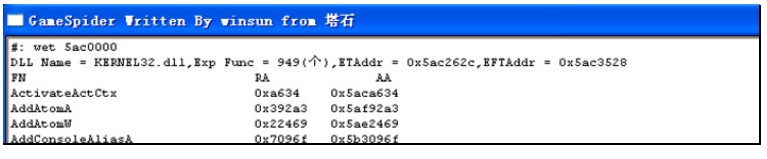


图 8-27　becf19844870.tmp 模块的导出属性

真正的 kernel32.dll 模块的内存块页面保护属性，如图 8-28 所示。

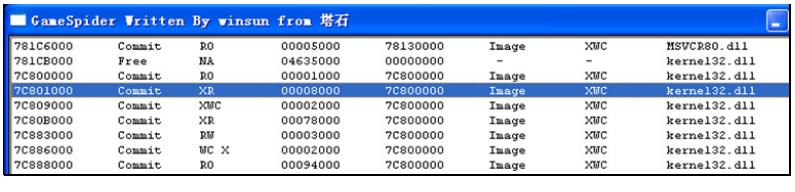


图 8-28　kernel32.dll 模块的内存块页面属性

对比图 8-26 和图 8-28：同一个模块在代码开始处的内存块中，页面保护属性是不同的，自加载模块少了可读属性，这样就使导出表不会被读取（因为导出表也在这个内存块中）。

8.3.2　ring0 保护方案

ring0 保护方案的工作流程大致如下。

（1）SSDT Hook。

本保护方案没有实施 SSDT Hook。

（2）SSDT Shadow Hook。

本保护方案对 NtUserSendInput 函数实施 Hook，主要目的是防止模拟按键，如图 8-29 所示。

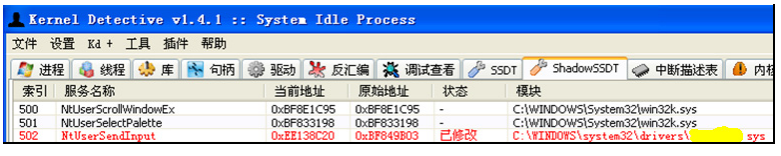


图 8-29　NtUserSendInput Hook

（3）Inline Hook，如图 8-30 所示。

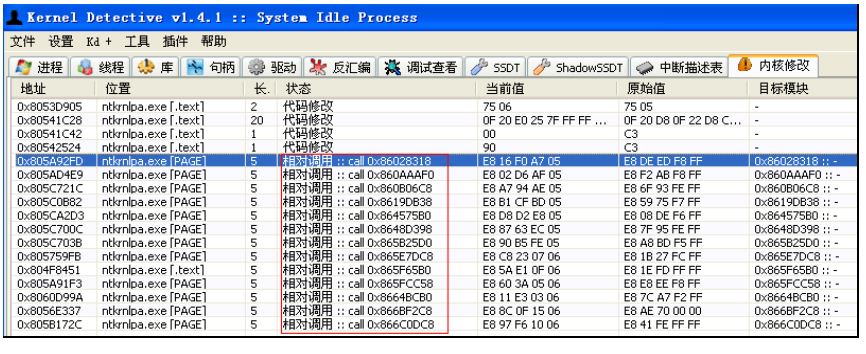


图 8-30　修改内核

KD 中显示的内核修改部分的前 4 行是游戏运行起来之前内核发生的改变，所以这应该不是游戏的行为导致的，我们可以直接从第 5 行开始分析。

要分析这些地址到底是做什么用的，可以先定位这些修改是在哪个内核函数中进行的，这有助于我们进行分析和判断。定位这些修改发生在哪个函数里的方法大致有以下两种。

- 把发生改变的地址记录下来，然后在“SSDT”界面观察与之最接近的地址在哪个函数中。
- 用 windbg 联调 VMWare，然后运行 ln 命令来查看，示例如下。

```
kd> ln 805a92fd
(805a92f6)   nt!NtWriteVirtualMemory+0x7
kd> ln 805ad4e9
(805ad4e2)   nt!NtProtectVirtualMemory+0x7
kd> ln 805c721c
(805c71f2)   nt!NtSetContextThread+0x2a
kd> ln 805c0b82
(805c0b78)   nt!NtOpenProcess+0xa
kd> ln 805ca2d3
(805ca2cc)   nt!PsSuspendThread+0x7
kd> ln 805c700c
(805c6fe2)   nt!NtGetContextThread+0x2a
kd> ln 805c703b
(805c6fe2)   nt!NtGetContextThread+0x59
kd> ln 805759fb
(805753ea)   nt!IopXxxControlFile+0x611
kd> ln 804f8451
(804f8438)   nt!KiAttachProcess+0x19
kd> ln 805a91f3
(805a91ec)   nt!NtReadVirtualMemory+0x7
kd> ln 8060d99a
(8060d8bc)   nt!NtQueryPerformanceCounter+0xde
kd> ln 8056e337
(8056e312)   nt!NtDeviceIoControlFile+0x25
kd> ln 805b172c
(805b1714)   nt!NtClose+0x18
```

上面的 13 个 Hook 集中反映了本保护模块的功能。虽然没有具体分析 Hook 之后做了什么，但是对 Windows 客户端安全有一定基础的读者应该很快能明白这个保护方案是如何进行防御的。

这 13 个 Hook 从功能上大致可以分成下面几类。

- 内存访问：NtWriteVirtualMemory 和 NtReadVirtualMemory，防御读写内存操作。
- 进程操作：KiAttachProcess 和 NtOpenProcess，防御打开或挂接进程操作。
- 线程操作：PsSuspendThread，防御悬挂关键线程操作。
- 硬件断点：NtGetContextThread 和 NtSetContextThread，检测和下硬件断点。
- 驱动通信：NtDeviceIoControlFile 和 IopXxxControlFile，保护 ring3 和 ring0 的通信。
- 时钟加速：NtQueryPerformanceCounter，防止加速。
- 页面属性：NtProtectVirtualMemory，保护自加载等重要模块的页面属性。
- 对象关闭：NtClose，保护关键对象不被恶意关闭。

对于如何绕过这些保护，方法大致有 3 种：第一，恢复所有的修改，但是这一般会导致蓝屏；第二，在游戏加载之前 Hook 这些保护点；第三，自加载一份内核。

8.4　本章小结

经过本章对一个游戏保护系统的简单分析，相信读者已经对分析游戏保护系统有了一定的认识——从定位保护模块，到分析保护模块，到制定绕过策略，整个过程充满了挑战。

虽然每款游戏的保护方案可能不尽相同，但是本章介绍的分析思路和工具还是可以派得上用场的。希望读者在今后的分析过程中，能够积累更多、更好的思路和工具，以提高分析效率和隐蔽性。

第 4 篇

射击游戏安全专题

第 9 章　射击游戏安全

第 9 章　射击游戏安全

本章主要围绕 FPS 游戏（First-Person Shooter Game，第一人称射击游戏）中一些常见的外挂功能进行解讲，包括自动开枪、反后坐力、透视等，同时会对开发 FPS 游戏外挂所需的脚本开发工具 AutoIt、易语言和 D3D9 进行介绍。下面就让我们一起开始 FPS 外挂之旅吧。

9.1　自动开枪

在 FPS 游戏中，谁瞄得准、按得快，谁的命中率和击毙率就高。但是，毕竟人的瞄准水平和按键频率是有限的，如果能用程序来模拟瞄准和按键的话，命中率和击毙率将提高许多。自动开枪机制就提供了这项功能。

自动开枪的实现需要两个步骤：一是自动瞄准；二是自动发射。在 FPS 游戏中，瞄准和发射分别对应于玩家移动鼠标和单击鼠标的动作。如果外挂能够模拟这两个动作，就可以实现无人工参与的自动开枪了。

幸运的是，Windows 平台支持模拟按键。

模拟按键是指模拟真实鼠标或键盘的按键行为。在 ring3 级，我们可以使用 SendMessage、PostMessage、SendInput 或 DirectInput（DirectX SDK）等 API 来实现模拟按键。在 ring0 级，我们可以构造 KEYBOARD_INPUT_DATA/MOUSE_INPUT_DATA，然后直接调用 KeyboardClassServiceCallback 和 MouseClassServiceCallback，或

者直接通过 out/in 指令来操作物理端口，从而实现模拟按键。

下面将在易语言和 AutoIt 的基础上来介绍模拟按键及其强大的功能。

9.1.1　易语言简介

易语言是一门计算机语言，支持中文编程，具体介绍参见 http://www.dywt.com.cn/。不知道从什么时候起，一些人开始使用易语言来编写外挂程序。可能有两个原因促使他们采用易语言：第一，易语言支持中文，API 等接口让人有见名知义的亲切感；第二，易语言是一门新语言，对逆向分析人员来说有点陌生，会增加一些分析难度。

如果读者对易语言不熟悉的话，可以阅读本章资源包中的相关文档。当然，网上也有很多文章，请读者自行搜索和阅读。

9.1.2　易语言版自动开枪外挂

在本节中，我们将开发一款基于取颜色和模拟按键功能的自动开枪外挂。该程序的外观如图 9-1 所示。

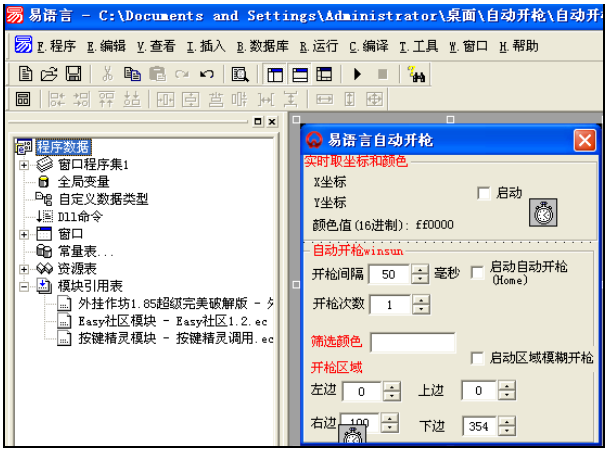


图 9-1　易语言版自动开枪外挂

易语言版自动开枪外挂有两个功能，一个是自动取坐标和取颜色，另一个就是自动开枪了。按下键盘上的【Home】键可以启动自动开枪功能，每隔 50 毫秒，程序

就会计算鼠标所在坐标的颜色是否满足筛选条件，如果满足，则发送模拟鼠标按键事件。

这款外挂主要使用了 3 个外部模块，我们可以从图 9–1 的模块引用表中看到具体的引用模块。在这些模块中，下面几个接口起到了核心作用。

- 取坐标，原型如下。

```
.子程序 按键精灵_获得鼠标坐标, , 公开
.参数 返回X, 整数型, 参考
.参数 返回Y, 整数型, 参考取坐标处的颜色
```

- 取颜色，原型如下。

```
.子程序 按键精灵_获得坐标点颜色, 文本型, 公开
.参数 横坐标, 整数型
.参数 纵坐标, 整数型
.参数 返回类型, 整数型, 可空, 1为十进制文本,2为十六进制文本 默认为2
```

- 模糊/精确查找某一区域所含颜色的坐标值，并移动鼠标至该坐标处。

“按键精灵_屏幕模糊查找颜色”函数的原型如下。

```
.子程序 按键精灵_屏幕模糊查找颜色, 逻辑型, 公开, 成功返回真,失败返回假
.参数 区域左边, 整数型
.参数 区域上边, 整数型
.参数 区域右边, 整数型
.参数 区域下边, 整数型
.参数 颜色值, 文本型, , 十六进制字符串
.参数 查找类型, 整数型, , 0为从上往下，从左往右找；1为从中心往外围找；2为从右下往左上
.参数 模糊参数, 小数型, , 取0.3~1之间的小数,数字越大相似越严格,推荐0.8
.参数 返回X, 短整数型, 参考, 如果未找到 为-1
.参数 返回Y, 短整数型, 参考, 如果未找到 为-1
```

“按键精灵_屏幕精确查找颜色”函数的原型如下。

```
.子程序 按键精灵_屏幕精确查找颜色, 逻辑型, 公开, 成功返回真,失败返回假
.参数 区域左边, 整数型
```

```
    .参数 区域上边，整数型
    .参数 区域右边，整数型
    .参数 区域下边，整数型
    .参数 颜色值，文本型，，十六进制字符串
    .参数 查找类型，整数型，，0 为从上往下，从左往右找；1 为从中心往外围找；2 为从右下往左上
    .参数 返回 X，短整数型，参考，如果未找到 为-1
    .参数 返回 Y，短整数型，参考，如果未找到 为-1
```

“按键精灵_屏幕模糊查找颜色”函数的具体实现如下。其中，“按键精灵_屏幕模糊查找颜色”函数将调用按键精灵里面封装的函数接口。

```
    子程序 按键精灵_屏幕模糊查找颜色，逻辑型，公开，成功返回真,失败返回假
    .参数 区域左边，整数型
    .参数 区域上边，整数型
    .参数 区域右边，整数型
    .参数 区域下边，整数型
    .参数 颜色值，文本型，，十六进制字符串
    .参数 查找类型，整数型，，0 为从上往下，从左往右找；1 为从中心往外围找；2 为从右下往左上
    .参数 模糊参数，小数型，，取 0.3~1 之间的小数,数字越大相似越严格,推荐 0.8
    .参数 返回 X，短整数型，参考，如果未找到 为-1
    .参数 返回 Y，短整数型，参考，如果未找到 为-1
    .局部变量 坐标值，整数型
    .局部变量 对象，对象
    .如果真 (查找类型 ＞ 2 或 查找类型 ＜ 0)
            返回 (假)
    .如果真结束
    对象.创建 (“QMDispatch.QMFunction”，)
    坐标值 ＝ 对象.数值方法 (“FindColor”，区域左边，区域上边，区域右边，区域下边，颜色值，查找类型，模糊参数)
    返回 X ＝ 取整 (坐标值 ÷ 8192)
    返回 Y ＝ 坐标值 % 8192
    对象.清除 ()
    .如果真 (返回 X ≤ 0 或 返回 Y ≤ 0)
```

```
        返回 X = -1
        返回 Y = -1
        返回 (假)
.如果真 结束
返回 (真)
```

下面从伪代码的角度来看看区域模糊开枪的逻辑，示例如下。

```
// 主线程
void main()
{
   // 创建开枪事件对象
   hStartEvent = CreateEvent(NonSigled);
   // 创建自动开枪线程
   CreateThread(AutoFireThread);
   while(1)
     {
if( GetAsyncKeyState( VK_HOME ) == TRUE )
      SetEvent( hStartEvent );          // 开启自动开枪
    else
         ResetEvent( hStartEvent );     // 关闭自动开枪
}
}

// 自动开枪线程
AutoFireThread()
{
while(1)
{
dwObject  =  Waitforsigleobject( hStartEvent, 无限等待);

   if ( dwObject == success )
       对象.创建 ("QMDispatch.QMFunction")
```

```
        // 从当前坐标周围开始查找与给定颜色值相似的坐标并返回坐标值
        坐标值 = 对象.数值方法 ("FindColor", 区域左边, 区域上边, 区域右
边, 区域下边, 颜色值, 查找类型, 模糊参数);
        X = 取整 (坐标值 ÷ 8192)
        Y = 坐标值 % 8192

        // 自动开枪
        Loop = 0;
        While( loop < 自动开枪的次数 )
            mouse_event( 左键按下, 0, 0, 0);        // 模拟鼠标左键按下
            mouse_event( 左键弹起, 0, 0, 0);        // 模拟鼠标左键弹起
            loop++;
       else
        Continue;
   }

   }
```

具体的易语言代码参见本章资源包。

9.2 反后坐力

所谓反后坐力，就是开枪之后防止枪的准星上扬，以减弱枪的抖动。在 FPS 游戏中，一旦枪支具有反后坐力，那么使用它的玩家下一次的瞄准和开枪动作就会比别人快一步。下面就让我们看看有什么办法可以减弱枪的后坐力。

9.2.1 平衡 Y 轴法

在反后坐力的实现方法中，有一种比较简单但有些粗糙的方式，那就是通过模拟按键的方式来平衡 Y 轴的上升变化，步骤如下。

（1）获取鼠标当前坐标 (X, Y) 。

（2）调用 mouse_event(…, X, Y +1, …) 函数模拟发送鼠标按键。

在这个方案里，每次开枪都要把 Y 轴的坐标加 1，然后调用 mouse_event() 函数发送模拟鼠标按键，其原理如图 9–2 所示。

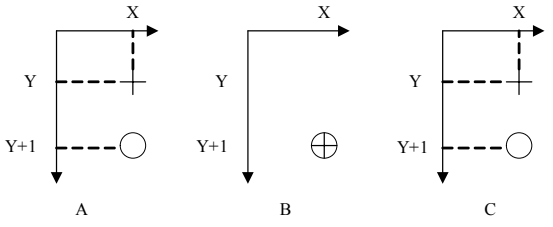


图 9-2　模拟开枪的过程

在图 9–2 中，加号代表准星，圆圈代表目标，它们都在以向右为 X 轴、向下为 Y 轴的屏幕坐标系中，状态 A、状态 B、状态 C 的含义分别如下。

- 状态 A：初始化时，准星置于目标之上。
- 状态 B：调用 GetCursorPos 获取准星当前坐标 (X, Y)，然后调用 mouse_event (…, X, Y+1, …) 函数击中目标。
- 状态 C：命中目标后，由于枪的后坐力影响，准星反弹至 (X, Y)。

经过上面的分析，如果在到达状态 C 之后继续调用 mouse_event(…, X, Y+1, …) 函数，那么理想状态下开枪的过程就会进入状态 B 到状态 C、状态 C 到状态 B 这样持续的循环，从而避免了后坐力的干扰。如果没有调用 mouse_event(…, X, Y+1, …) 函数，那么在到达状态 B 之后，由于后坐力的影响，将到达状态 C，这时必须手动调整准星——这明显降低了开枪的速度，也降低了命中率。

在实际应用中，开枪之后，后坐力反弹的距离可能不止 1 像素，这就需要测试开枪前和开枪后后坐力反弹的距离 Z，然后在平衡 Y 轴中调用 mouse_event(…, X, Y+Z, …) 函数。

9.2.2　AutoIt 脚本法

本节通过 AutoIt 来实现一个基于平衡 Y 轴的反后坐力 Demo。使用 AutoIt 可以快速编写基于脚本的代码。下面就让我们先了解一下 AutoIt 这款软件吧。

AutoIt 是一个使用类似 Basic 脚本语言的免费软件，用于在 Windows GUI（图形

用户界面）中进行自动化操作。它利用模拟键盘按键、鼠标移动和窗口/控件的组合来完成自动化任务，而这是其他语言不可能做到或者没有可靠方法能够实现的（例如 VBScript 和 SendKeys）。

AutoIt 最初是为一台 PC 对数千台 PC 进行配置的批量处理而设计的，不过随着 AutoIt 3 的出现，它也适用于家庭自动化和编写用以完成重复性任务的脚本。

AutoIt 可以完成的工作如下。

- 运行 Windows 和 DOS 程序。
- 模拟击键动作（支持大多数键盘布局）。
- 模拟鼠标移动和点击操作。
- 对窗口进行移动、调整大小等操作。
- 直接与窗口的控件交互（设置/获取文本、移动、关闭等）。
- 配合剪贴板进行剪切/粘贴文本操作。
- 对注册表进行操作。

AutoIt 的脚本文件都是可执行文件，其脚本文件扩展名为 .au3。因为 AutoIt 编写的代码是解释执行的，其运行依赖一个解释器，即由这个解释器来翻译和解释脚本的每条命令（或者说代码），然后执行相应的操作，所以我们必须在机器上安装 AutoIt 软件。如果不进行严格的定义，HTML 和 Java 都可以被认为是解释性语言。AutoIt 3 的主程序 AutoIt3.exe 就是它的解释器，上面提到脚本能够转换成可脱离相应解释器独立运行的可执行文件，而且，我们可以使用相应的工具把它们还原成脚本文件。由此我们完全可以这样理解：脚本代码是被“压缩”到这个可执行文件中的，解释器也在里面，在运行可执行文件时，实际上是先“解压”脚本代码，然后运行解释器并解释该脚本。

具体的脚本代码参见本章资源包中的脚本文件 winsun.au3。该脚本文件经过 SciTE4AutoIt3 编辑器的编译，可以在 winsun.au3 的对应目录下生成 winsun.exe 文件。

从脚本文件 winsun.au3 中取出核心功能代码，示例如下。

```
    While 1
     If _IsPressed('1b') = 1 Then Exit;1b is ESC ;  //判断【Esc】键是否
按下
     If _IsPressed('01') = 1 Then ;   //判断鼠标左键是否按下
```

```
        $MousePos = MouseGetPos( ) ;   //获取当前鼠标坐标
    MouseMove( $MousePos[0], $MousePos[1] + 1, 1000) ;   //当前鼠标 Y 坐
标轴上的值加 1 后,模拟鼠标按键
    EndIf
      Sleep(100)
    Wend
    ;下面的 Func_IsPressed 函数用于判断某个键是否按下
    Func _IsPressed($hexKey)
    ; $hexKey must be the value of one of the keys.
    ; _IsPressed will return 0 if the key is not pressed, 1 if it is.
    ; $hexKey should entered as a string, don't forget the quotes!
    ; (yeah, layer. This is for you)
    ;virtual-key code
    ;VK_LBUTTON (01)
     Local $aR, $bO

      $hexKey = '0x' & $hexKey
      $aR = DllCall("user32", "int", "GetAsyncKeyState", "int", $hexKey)
      If Not @error And BitAND($aR[0], 0x8000) = 0x8000 Then
         $bO = 1
      Else
         $bO = 0
      EndIf
    Return $bO
    EndFunc  ;==>_IsPressed
```

winsun.exe 的运行界面如图 9-3 所示。

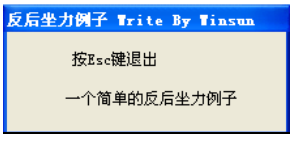


图 9-3　winsun.exe 运行界面

按下【Esc】键，winsun.exe 将会退出。根据脚本文件 winsun.au3 的代码，当用户按下鼠标左键时，程序会通过鼠标移动函数重新调整鼠标的位置。

9.3 DirectX Hack

本节主要介绍 DirectX Hack，而非讨论如何用 DirectX 来编写游戏。所以，对 DirectX 的介绍也主要是为了让读者简单了解 COM 对象的基本内存对象布局以及如何使用 DirectX 实现简单的绘图功能，从而为后续的 D3D9 Hook 做铺垫。

9.3.1 DirectX 简介

DirectX（Direct Extension，简称 DX）是由微软公司开发的多媒体编程接口，它采用 C++ 语言编写，遵循 COM 思想。DX 的组件比较多，主要分成 4 个部分，分别是显示部分、声音部分、输入部分和网络部分。

下面让我们通过一个简单的例子来了解一下 COM（这个例子来自《Windows 游戏编程大师技巧》一书的第 5 章）。

```
// 定义 IX 接口
interface IX: IUnknown
{

virtual void __stdcall fx(void)=0;

};

// 定义 IY 接口
interface IY: IUnknown
{

virtual void __stdcall fy(void)=0;

};
// 定义 COM 对象
class CCOM_OBJECT :    public IX,
                  public IY
```

```
    {
    public:
    CCOM_OBJECT() : ref_count(0) {}
       ~CCOM_OBJECT() {}

    private:

    virtual HRESULT __stdcall QueryInterface(const IID &iid, void
**iface);
    virtual ULONG __stdcall AddRef();
    virtual ULONG __stdcall Release();

    virtual void __stdcall fx(void) {cout << "Function fx has been
called." << endl; }
    virtual void __stdcall fy(void) {cout << "Function fy has been
called." << endl; }

    int ref_count;

    };
    // 获取目标接口指针
    HRESULT __stdcall CCOM_OBJECT::QueryInterface(const IID &iid, void
**iface)
    {

    // 请求 IUnknown 接口
    if (iid==IID_IUnknown)
       {
       cout << "Requesting IUnknown interface" << endl;
       *iface = (IX*)this;

       }

    // 请求 IX 接口
```

```
    if (iid==IID_IX)
       {
       cout << "Requesting IX interface" << endl;
       *iface = (IX*)this;

       }
    else  // 请求 IY 接口
    if (iid==IID_IY)
       {
       cout << "Requesting IY interface" << endl;
       *iface = (IY*)this;

       }
    else
       { // 找不到对应的接口
       cout << "Requesting unknown interaface!" << endl;
       *iface = NULL;
       return(E_NOINTERFACE);
       }

    // 如果一切正常，就把指针指向 IUnknown 并调用 addref()函数
    ((IUnknown *)(*iface))->AddRef();

    return(S_OK);

    }

    // 增加引用计数
    ULONG __stdcall CCOM_OBJECT::AddRef()
    {
    // 增加引用计数
    cout << "Adding a reference" << endl;
    return(++ref_count);

    }
```

```
    // 减少引用计数
    ULONG __stdcall CCOM_OBJECT::Release()
    {
    // 减少引用计数
    cout << "Deleting a reference" << endl;
    if (--ref_count==0)
        {
        delete this;
        return(0);
        }
    else
        return(ref_count);

    }

    // 创建 COM 对象
    IUnknown *CoCreateInstance(void)
    {
    // 这是 CoCreateInstance 的实现，它将创建一个 COM 对象实例

    IUnknown *comm_obj = (IX *)new(CCOM_OBJECT);

    cout << "Creating Comm object" << endl;

    // 更新引用计数
    comm_obj->AddRef();

    return(comm_obj);

    }

    ////////////////////////////////////////////////////////////////
////////////////////////////////
```

```
void main(void)
{

// 创建 COM 对象
IUnknown *punknown = CoCreateInstance();

// 初始化 IX 和 IY 指针为 NULL
IX *pix=NULL;
IY *piy=NULL;

// 从 COM 对象中查询 IX 接口
punknown->QueryInterface(IID_IX, (void **)&pix);

// 调用 IX 接口提供的 fx()函数
pix->fx();

// 减少 IX 接口引用计数
pix->Release();

// 从 COM 对象中查询 IY 接口
punknown->QueryInterface(IID_IY, (void **)&piy);

// 调用 IY 接口提供的 fy()函数
piy->fy();

// 减少 IY 接口引用计数
piy->Release();

// 减少 COM 对象引用计数
punknown->Release();

}
```

上面这段程序是一个简单的 COM 编程示例，详细代码和对应程序参见本章资源包中的 com.cpp 和 com.exe 这两个文件。没有接触过 COM 编程的读者，可能会对上

面的代码风格比较陌生。不过不用担心，下面就让我们通过 IDA 来静态逆向分析 com.exe 程序，看看 COM 对象是如何在内存中管理数据和函数的。

CCOM_OBJECT 的内存对象布局图如图 9-4 所示，其中 CCOM_OBJECT 占据了 12 字节内存，第一个 4 字节的 VTBPIX 是指向 IX 接口的虚表指针，第二个 4 字节的 VTBPIY 是指向 IY 接口的虚表指针，第三个 4 字节的 ref_count 是引用计数。IX 接口指针是 ECX 所指向的 CCOM_OBJECT 对象地址，IY 接口指针是 ECX+4 所指向的 CCOM_OBJECT 对象地址偏移 4 字节处。

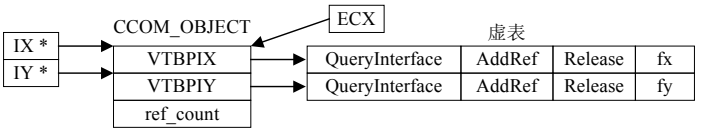


图 9-4　CCOM_OBJECT 内存对象布局

根据以上逆向分析，我们可以获得如下信息。

- “IUnknown *punknown = CoCreateInstance();”语句：此时的 punknown 就等于 ECX，指向一个占据 12 字节的 CCOM_OBJECT 对象。
- “punknown->QueryInterface(IID_IX, (void **)&pix);”语句：pix 也等于 ECX，指向一个占据 12 字节的 CCOM_OBJECT 对象。
- “pix->fx();”语句：上面对 fx() 函数的调用就等于调用“(fx*)*((PDWORD) VTBPIX+3)”语句。
- “pix->Release();”语句：上面对 Release 的调用就等于调用“(Release*)* ((PDWORD)VTBPIX+2)”语句。
- “punknown->QueryInterface(IID_IY, (void **)&piy);”语句：此时查询 piy 的结果就等于“ECX+4”，指向 CCOM_OBJECT 对象偏移 4 字节处。
- “piy->fy();”语句：上面对 fy() 函数的调用就等于调用“(fy*)*((PDWORD) VTBPIY+3)”语句。
- “piy->Release();”语句：上面对 Release 的调用就等于调用“(Release*)* ((PDWORD)VTBPIY+2)”语句。
- “punknown->Release();”语句：上面对 Release 的调用就等于调用“(Release*)* ((PDWORD)(VTBPIX)+2)”语句。

通过对这个 COM 例子和内存对象布局的分析，相信读者已经对 COM 有了一定的认识。下面让我们继续了解如何用基于 COM 思想的 D3D9 来绘制简单的图形。

9.3.2 用 Direct3D 绘制图形

Direct3D 是 DirectX 组件中的 3D 图形部分，大量运用在各种 3D 图形游戏中，如《穿越火线》、《反恐精英》等。在使用 Direct3D 之前，用户必须安装 DirectX SDK 的对应版本。本节使用的 SDK 是 9.0 版本，所以笔者称之为 D3D9。

作为一个完整的 D3D9 程序，其基本步骤大致有如下 3 步。

（1）初始化 Direct3D，示例如下。

```
      // 创建 Direct3D 对象，并获取接口 IDirect3D9 的指针
      // 我们将通过 IDirect3D9 指针来操作 Direct3D 这个 COM 对象
      g_pD3D = ::Direct3dCreate9(D3D_SDK_VERSION );
     // 调用 IDirect3D9::CreateDevice 来创建设备对象，并获取接口 IDirect3D
Device9 的指针
     // 我们将通过该指针操作设备对象
     g_pD3D->CreateDevice(
   D3DADAPTER_DEFAULT,    // 使用默认的显卡适配器
   D3DDEVTYPE_HAL,        // 请求使用硬件抽象层（HAL）
   hWnd,   // 窗口句柄
          D3DCREATE_SOFTWARE_VERTEXPROCESSING,    // 软件处理顶点
          &d3dpp,  // 创建的参数
          &g_pD3DDevice   // 创建的 IDirect3DDevice9 设备指针
   );
```

（2）初始化完成后，就可以开始执行渲染操作，即前面所说的绘图了。Direct3D 渲染一幅图像时，先在后备缓存区渲染，渲染结束后交换到当前缓存区，然后在后备缓存区渲染后续图像，渲染过程如图 9–5 所示，示例代码如下。

```
   // 渲染前
   ce->Clear(   0,   NULL,   D3DCLEAR_TARGET   |   D3DCLEAR_ZBUFFER,
D3DCOLOR_XRGB(0,0,255), 1.0f, 0 );
```

```
// 开始绘制场景
g_pD3DDevice->BeginScene();

// 省略具体渲染代码，因渲染方式和图形的不同而不同

// 结束绘制场景
g_pD3DDevice->EndScene();

// 交换当前和后备缓存区，刷新窗口
g_pD3DDevice->Present( NULL, NULL, NULL, NULL );
```

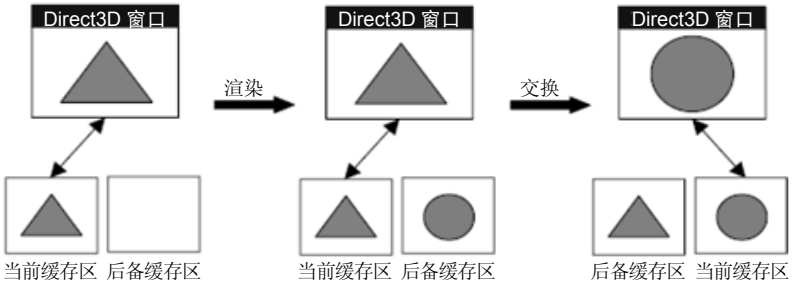


图 9-5　Direct3D 渲染过程

具体的图形渲染代码，读者可以参考本章资源包中 D3D9Hack.cpp 文件里的 Render 函数。

在渲染图形之前，要先设置顶点缓存，再调用 DrawPrimitive 或 DrawIndexed Primitive 来绘制图形。DrawPrimitive 和 DrawIndexedPrimitive 的区别在于前者的顶点缓存区顺序列出了图形中所有的顶点，而后者则设置了一组顶点和一组索引，索引与顶点之间利用编号建立联系，从而避免了顶点重复定义带来的内存浪费问题，提高了效率。

（3）结束 Direct3D，释放资源。Direct3D 程序退出前，需要释放相应的资源，这时必须调用对应接口的 Release 函数。

下面的 D3D9Hack.exe 就是一个根据上面的步骤基于 DX9 编写的简单绘图程序，如图 9–6 所示。读者可以尝试在 D3D9 绘图里尝试绘制各种图形。

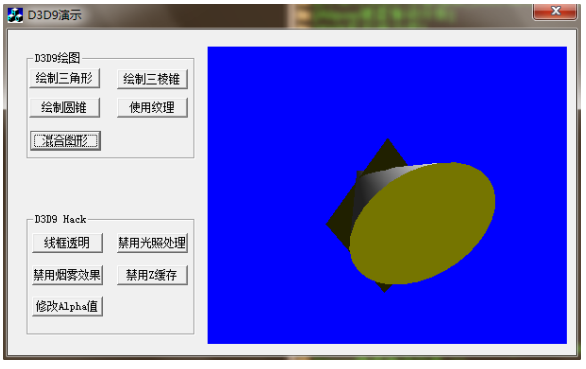


图 9-6　DX9 绘图及 Hack 界面

在本节中，我们不仅了解了 COM 编程的基本要素和 COM 对象的基本内存布局，还了解了 DX 绘图的基本过程。下面就让我们看看 DX 中有哪些值得 Hack 的功能点。

9.3.3　D3D9 的 Hack 点

根据对 DX 绘图过程的分析可以发现，渲染方式对图形成型起着至关重要的作用，而渲染方式的设置来自 IDirect3DDevice9 设备的 SetRenderState 方法，所以，大多数 D3D9 Hack 功能都需要调用 SetRenderState 方法来实现。

下面列举一些外挂中常见的 D3D9 Hack 方法。

- 线框透明：通过调用 SetRenderState 方法修改 IDirect3DDevice9 设备的填充模式，在渲染时忽略图形表面的材质和纹理等，只绘制带顶点的线框，具体调用方式如下。D3D9Hack.exe 程序绘制的圆锥体经过线框透明设置后，如图 9–7 所示。

```
g_pD3DDevice ->SetRenderState(D3DRS_FILLMODE, D3DFILL_WIREFRAME);
```

- 禁用光照：处理后，光线对物体表面的材质不起作用，物体表面将以高亮显示。在 DX 游戏中，这一功能可以高亮显示要关注的物体对象，具体调用方式如下。

```
g_pD3DDevice ->SetRenderState(D3DRS_LIGHTING, FALSE);
```

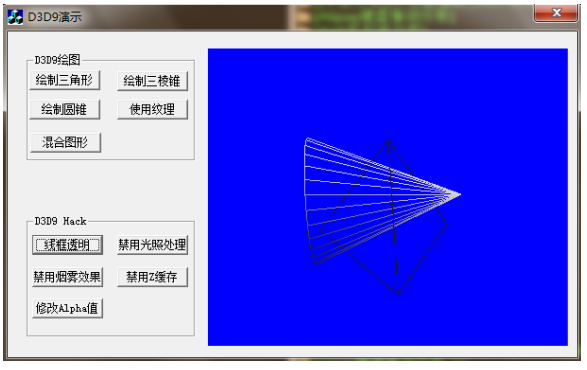


图 9-7　线框透明

➢ 禁用烟雾效果：处理后，绘制场景中的烟雾效果将会消失，这在 FPS 游戏的屏蔽烟雾弹功能中有显著作用，具体调用方式如下。

```
g_pD3DDevice ->SetRenderState(D3DRS_FOGENABLE, FALSE);
```

➢ 禁用 Z 缓存：在 D3D9 中，可以使用深度缓存区来进行消隐处理（隐藏面消除），以确保实体被遮挡的部分不被显示。Z 缓存是一种常用的深度缓存，它以 Z 坐标作为判断深度（远近）的依据而得名。所以，在 D3D9 中，我们只要禁用 Z 缓存，被挡住的物体就会呈现在面前，到达透视的效果。禁用 Z 缓存的具体调用方式如下。D3D9Hack.exe 程序绘制的混合图形禁用 Z 缓存后的效果如图 9–8 所示。

```
g_pD3DDevice ->SetRenderState(D3DRS_ZENABLE, FALSE);
```

➢ 修改材质的 Alpha 值以实现透明效果。

Direct3D 渲染一个图形元素时，总是根据图形元素的材质和有关的光线信息来产生图形元素的颜色。如果程序中的纹理融合有效，Direct3D 就必须将一个或多个纹理的纹理像素颜色与图形元素的当前颜色进行融合。

Direct3D 用下面的公式来决定一个图形元素中每个像素最终的颜色。

```
FinalColor = TexelColor * SourceBlendFactor + PixelColor * DestBlend
Factor
```

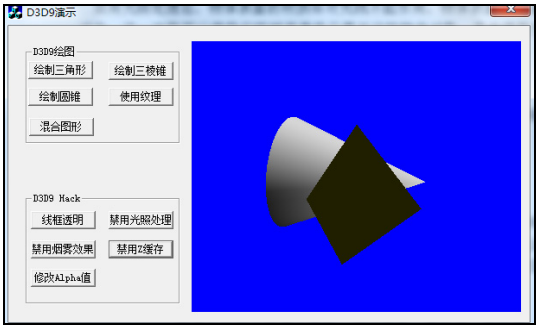


图 9-8　禁用 Z 缓存

在上面的公式中：FinalColor 是最后输出到目标渲染表面的像素颜色；TexelColor 表示对应于当前像素的纹理像素的颜色；SourceBlendFactor 是一个通过计算得到的值，用于决定将纹理像素颜色的百分之多少应用于最后的颜色；PixelColor 是图形元素中当前像素的颜色；DestBlendFactor 用于决定将当前像素颜色的百分之多少用于最后的颜色。SourceBlendFactor 和 DestBlendFactor 的取值范围为 0.0 ~ 1.0。

从上式中我们可以知道，当 SourceBlendFactor 为 0.0 并且 DestBlendFactor 为 1.0 时，纹理是完全透明的。如果它们是其他的值，那么得到的纹理就会有不同程度的透明。纹理中的每个纹理像素都有一个红、绿、蓝色值。

默认情况下，Direct3D 将 Alpha 值作为 SourceBlendFactor。因此，可以通过设置纹理的 Alpha 值来控制透明度。

D3DRENDERSTATE_SRCBLEND 和 D3DRENDERSTATE_DESTBLEND 枚举值可以用来控制融合因子。要使用它们，就要引用 IDirect3DDevice3::SetRenderState 方法，并将 D3DRENDERSTATE_SRCBLEND 或 D3DRENDERSTATE_DESTBLEND 作为它的第一个参数，将 D3DBLEND 枚举类型的一个成员作为它的第二个参数，关键代码如下。

```
        // 开启 Alpha 融合
        g_pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, true);
        // 设置融合因子，使 Alpha 组件决定透明度
        g_pD3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
        g_pD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_
xINVSRCALPHA);
```

```
        // 设置为半透明
        g_pD3DDevice->GetMaterial(&stMaterial);
        stMaterial.Diffuse.a    = 0.5f;
    g_pD3DDevice->SetMaterial(&stMaterial);
```

具体代码参见本章资源包中该程序的修改 Alpha 值按钮所对应的代码。

虽然本节对 D3D9 Hack 功能点进行了详细的分析，但是要使这些 Hack 功能得到执行，必须在游戏的正常执行逻辑里插入我们的 Hook 代码，以便在特定的时机实现外挂功能。第 9.3.4 节将介绍一种能更好地 Hook 以防止被反外挂系统检测到的方法。

9.3.4　D3D9 Hook

根据第 9.3.2 节中关于 Direct3D 绘图过程的描述，我们可以知道，Direct3D 的核心功能集中在 IDirect3DDevice9 接口中。所以，只要能 Hook 其中的 EndScene() 函数、DrawPrimitive() 函数或 DrawIndexedPrimitive() 函数，就能感知绘图操作，从而实现 Hack 功能。

在进行 Hook 之前，让我们先来看看 Direct3D 中关键对象的内存布局，如图 9–9 所示。

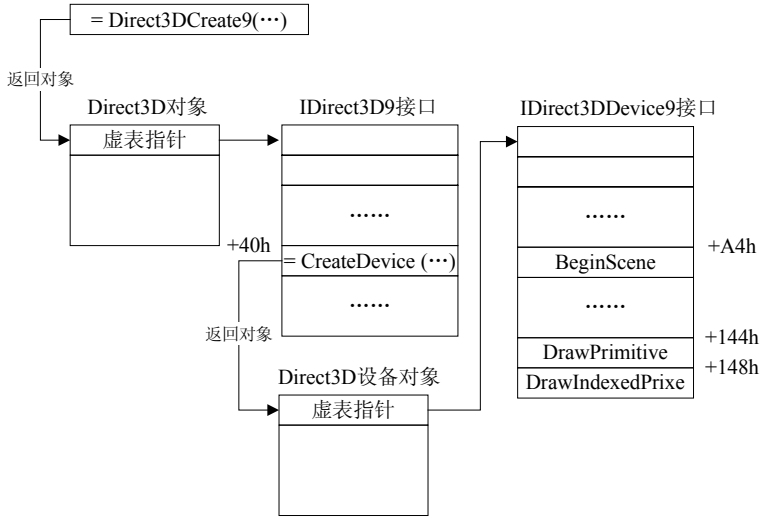


图 9-9　Direct3D 对象内存布局

要使用 Direct3D 提供的接口进行绘图等操作，需要下面 3 个步骤。

（1）调用 Direct3dCreate9() 函数创建 Direct3D 对象，获取 IDirect3D9 接口。

（2）调用 IDirect3D9 接口中偏移 40h 处的 CreateDevice() 函数来创建 Direct3D 设备对象，获取 IDirect3DDevice9 接口。

（3）调用 IDirect3DDevice9 接口中的函数实现绘图等操作。

我们的 Hook 目标就是 IDirect3DDevice9 接口中的函数。要 Hook 这个接口中的函数，必须解决以下两个问题。

- 如何定位 IDirect3DDevice9 接口的内存地址。
- 如何 Hook IDirect3DDevice9 接口中的函数并躲避检测。

显然，根据图 9-9 中箭头的指向，我们可以反推出：只要 Hook 了 Direct3DCreate9 函数，就能根据箭头的指向获取 IDirect3DCreate9 接口的地址。

而对于一个函数的 Hook，我们可以采用以下两种方式进行。

- 修改被 Hook 函数的头或尾（修改代码容易被 Hash 校验查出）。
- 修改 IDirect3DDevice9 接口中存放被 Hook 函数的位置中的值（修改数据容易被校验是否处在某个合法模块的地址区间）。

为了避免被反外挂系统检测到，下面将采用以上第二种方式来 Hook（类似 IAT Hook）。不过，在 Hook 之前，必须在 D3D9 模块的代码段区间寻找一段代码空隙，然后把空隙变成一个具有跳转功能的指令块，让其跳转到 Hook 后的功能函数处，最后把空隙的地址填入 IDirect3DDevice9 接口内对应的插槽中，从而实现 Hook。

当然，寻找代码空隙和 Hook 等都是在一个完整的模块中完成的，我们可以采用之前讨论的注入方法来实现完整模块的注入，推荐的注入方式有 ComRes 注入、输入法注入和 LSP 注入等。

注入之后，Hook 的流程如下。

（1）IAT Hook kernel32.dll 模块中的 LoadLibraryA 函数，感知 D3D9 模块的动态加载时机。

（2）EAT Hook d3d9.dll 模块中的 Direct3DCreate9 函数，感知 Direct3D 对象的创建时机。

（3）Hook IDirect3D9 接口中的 CreateDevice 函数，感知 Direct3D 设备对象的创建时机。

（4）Hook IDirect3DDevice9 接口中的 DrawIndexedPrimitive 函数，感知绘图时机。

上述流程中的第 1 步涉及 IAT Hook，其核心代码如下。

```
    // pszIndexModuleName: 待搜索的被导入模块的名字
    // HookAPIAddr: 待搜索的被导入函数的地址
    // lpNewFunc: Hook 后新函数的地址
    // hSearcModule: 搜索的目标模块
    BOOL    IATHookForOneModule(CHAR    *pszIndexModuleName,    PROC
HookAPIAddr, PROC lpNewFunc, HMODULE hSearcModule)
    {

        // 判断搜索的目标模块是否是合法的 PE 模块
        pDosHd = ( PIMAGE_DOS_HEADER )hSearcModule;
        if ( pDosHd->e_magic != IMAGE_DOS_SIGNATURE )
            return false;
        pNtHd = ( PIMAGE_NT_HEADERS )( (PBYTE)pDosHd + pDosHd->e_lfanew);
        if ( pNtHd->Signature != IMAGE_NT_SIGNATURE )
            return false;

        // 判断搜索的目标模块是否含有导入表
        if ( pNtHd->OptionalHeader.DataDirectory[1].Size == 0 ||
            pNtHd->OptionalHeader.DataDirectory[1].VirtualAddress == 0)
          return false;

    // 获取导入表的虚拟地址
    pImpTable  =  (PIMAGE_IMPORT_DESCRIPTOR)(  (PBYTE)hSearcModule  +
pNtHd->OptionalHeader.DataDirectory[1].VirtualAddress );

        // 搜索定位导入表中目标导入模块的位置
        while( pImpTable->Characteristics != 0 )
        {
                pDstName  = (PCHAR)((PBYTE)hSearcModule + pImpTable->
Name);
```

```
                if ( stricmp(pszIndexModuleName, pDstName) == 0 )
                {
                   bSuccess = true;
                     break;
                }
                pImpTable++;
        }

    // 定位搜索模块中目标模块导入表的 IMAGE_THUNK_DATA 地址
    PIMAGE_THUNK_DATA  pThunk = (PIMAGE_THUNK_DATA)(pImpTable->First
Thunk + (PBYTE)hSearcModule);
    for(;pThunk->u1.Function;pThunk++)
    {
            PROC *ppfn = (PROC *)&pThunk->u1.Function;
            pOrigFunc = (PROC)pThunk->u1.Function;
            // 定位存放目标函数的地址
            if (*ppfn == HookAPIAddr)
            {
                    MEMORY_BASIC_INFORMATION mbi;
                    ZeroMemory(&mbi,
sizeof(MEMORY_BASIC_INFORMATION));
                    // 将 ppfn 所在页面的保护属性改为可读写
                    VirtualQuery(ppfn,    &mbi,    sizeof(MEMORY_BASIC_
INFORMATION));
       VirtualProtect(mbi.BaseAddress,      mbi.RegionSize,      PAGE_
READWRITE,&mbi.Protect);
                    // 写入新函数的地址，实现 IAT Hook
                    *ppfn = *lpNewFunc;
                    DWORD dwOldProt;
                    // 恢复 ppfn 所在页面的保护属性
                    VirtualProtect(mbi.BaseAddress,    mbi.RegionSize,
mbi.Protect,&dwOldProt);
                    return TRUE;
```

```
            }
    }
    }
```

上述流程中的第 2 步涉及 EAT Hook，其核心代码如下。

```
    // pszIndexModuleName：待搜索的导出模块的名称
    // HookAPIAddr：待 Hook 函数的地址
    // lpNewFunc：新函数的地址
    DWORD EATHook(CHAR *pszIndexModuleName, PROC HookAPIAddr, PROC
lpNewFunc)
    {

        // 获取待搜索模块的基地址
        HMODULE hDstModule = GetModuleHandle(pszIndexModuleName);
        __try
        {
            // 判断待搜索模块是否是合法的 PE 模块
            pDosHd = ( PIMAGE_DOS_HEADER )hDstModule;
            if ( pDosHd->e_magic != IMAGE_DOS_SIGNATURE )
            {
                OutputDebugString(" [-] 所指向的模块 PE 格式非法!\n");
                return bSuccess;
            }
            pNtHd = ( PIMAGE_NT_HEADERS )( (PBYTE)pDosHd + pDosHd->
e_lfanew);
            if ( pNtHd->Signature != IMAGE_NT_SIGNATURE )
            {
                OutputDebugString(" [-] 0x%x 所指向的模块 PE 格式非法!\n");
                return bSuccess;
            }

            // 定位目标模块的导出表地址
            if ( pNtHd->OptionalHeader.DataDirectory[0].Size == 0 ||
```

```
                pNtHd->OptionalHeader.DataDirectory[0].VirtualAddress
== 0)
            {
                OutputDebugString("have no export table!");
                return FALSE;
            }
        pExpTable = (PIMAGE_EXPORT_DIRECTORY)( (PBYTE)hDstModule +
pNtHd->OptionalHeader.DataDirectory[0].VirtualAddress );
    // 获取存放导出函数地址的表基地址
    PDWORD pdwFuncs = (PDWORD)((PBYTE)hDstModule + pExpTable->
AddressOfFunctions);
            // 遍历导出地址表，匹配待搜索函数所在地址表的槽位置
            for(DWORD dwLoop = 0; dwLoop<pExpTable->NumberOfFunctions;
dwLoop++)
            {
                if ( (DWORD)HookAPIAddr == ((DWORD)hDstModule +
pdwFuncs[dwLoop]))
                {
                    bSuccess = TRUE;
                    break;
                }
            }

            if (!bSuccess)
            {
                OutputDebugString("EATHook cannot find dest func!");
                return FALSE;
            }

           MEMORY_BASIC_INFORMATION mbi;
            ZeroMemory(&mbi, sizeof(MEMORY_BASIC_INFORMATION));
    // 查询导出地址表的页面保护属性
    VirtualQuery(&pdwFuncs[dwLoop], &mbi,
```

```
    sizeof(MEMORY_BASIC_INFORMATION));
        // 将导出地址表的页面属性改为可读写
        VirtualProtect(mbi.BaseAddress, mbi.RegionSize, PAGE_READWRITE,
&mbi.Protect);
            // 保存导出表中原始函数的地址
            dwOldAddr = pdwFuncs[dwLoop] + (DWORD)hDstModule;
            // 替换导出地址表中的原始函数为新函数地址
            pdwFuncs[dwLoop] = (DWORD)lpNewFunc - (DWORD)hDstModule;
            DWORD dwOldProt;
            // 恢复导出地址表的页面保护属性
            VirtualProtect(mbi.BaseAddress, mbi.RegionSize, mbi.Protect,
&dwOldProt);
            return TRUE;

        }
        __except( EXCEPTION_EXECUTE_HANDLER )
        {
            OutputDebugString("[!]    IATHookForOneModulew    Exception
Exit!\n");
            return FALSE;
        }
    }
```

上述流程中的第 3 步和第 4 步所涉及的操作是一致的，其核心实现代码如下。

```
    // pdwOldAddr: 存放虚函数的地址槽
    // pfnMyFuncAddr: Hook 后的函数
    // return: 返回原始函数的地址
    DWORD VTBHook(PDWORD pdwOldAddr, PROC pfnMyFuncAddr)
    {
        MEMORY_BASIC_INFORMATION mbi;
        DWORD dwOldProt;
        DWORD dwOldFuncAddr;
        ZeroMemory(&mbi, sizeof(MEMORY_BASIC_INFORMATION));
```

```
        VirtualQuery(pdwOldAddr, &mbi, sizeof(MEMORY_BASIC_INFORMATION));
        VirtualProtect(mbi.BaseAddress, mbi.RegionSize, PAGE_READWRITE,
&mbi.Protect);
        dwOldFuncAddr = *pdwOldAddr;
        *pdwOldAddr = (DWORD)pfnMyFuncAddr;
        VirtualProtect(mbi.BaseAddress,  mbi.RegionSize,  mbi.Protect,
&dwOldProt);
        return dwOldFuncAddr;
    }
```

为了演示 D3D9 Hook 的效果，本章的资源包中提供了两份工程代码，下面分别进行介绍。

- D3D9HackNew.exe 启动后，界面如图 9-10 所示。使用工具把 FindFreeMem.dll 注入进程，先单击“加载 D3D9”按钮，再单击“绘制三棱锥”等绘图按钮，就可以看到线框透明的效果了。

图 9-10　D3D9 演示程序

- FindFreeMem 是实现 D3D9 Hook 的核心模块，读者可以从 DllMain() 函数开始跟踪学习。

因为在 D3D9 模块加载前，FindFreeMem 这个 DLL 就已经注入演示程序，并进行了一系列的 Hook 操作，所以，单击“加载 D3D9”按钮的时候，就能获取 Direct3D 设备对象的虚表地址并进行 Hook。

9.4 本章小结

本章主要讨论第一人称射击游戏的安全问题，包括自动开枪、反后坐力以及各种常见的 DX Hack 功能。DX Hack 有一个比较难解决的问题是不能在 Direct3D 设备对象创建之后再去 Hook，而必须在 DX 加载之前就 Hook LoadLibrary。

第 5 篇

外挂检测技术篇

第 10 章　外挂的检测方法

本章是全书的最后一章。在本章中，让我们一起从外挂的特征入手，看看应该如何设计有效的思路来检测和防御外挂，保护游戏的安全。

在外挂行为检测方面，修改数据的行为最难定位，因为不仅外挂会修改数据，游戏本身也有可能修改数据，即对于存放游戏数据的内存来说，不仅数据内容可能会变，有些游戏还可能会动态改变存放数据的地址。所以，要通过差异分析的思想来定位外挂修改数据的行为是不可靠的。笔者曾引入一套线程转移和消息分流机制来定位外挂修改数据的行为，虽然不能做到百分之百准确，但是相信可以引发读者的思考，从而设计出更好的解决方案。

其实，从外挂对游戏进程的利用的角度来看，外挂的行为无非有如下几种。

- 修改游戏代码：属于篡改行为。
- Call 游戏函数：属于未授权访问行为。
- 修改游戏的关键数据：属于篡改行为。

针对上面每一种外挂的功能，下面先让我们大致了解一下如何去设计分析方法来分析它们。

10.1　代码篡改检测

事实上，修改游戏代码是很容易被反外挂系统检测到的——只要给代码段加一个校验功能，基本上就能检测出来。不过，安全原本是一个对抗过程，外挂为了躲避

检测，可以通过修改检测代码本身来绕过检测。这里主要讨论反外挂分析人员应该采用什么方法来快速定位被修改代码的地址。在分析方法之前，让我们直观地了解一下修改代码对游戏造成的影响。

多人角色扮演格斗游戏中常有一种外挂，会提供一种叫做“无敌”的功能。“无敌”的意思是当怪物攻击角色时角色不掉血。这种“无敌”往往是通过某种方式让怪物攻击角色时绕过碰撞检测的过程。假设我们要对某款游戏代码的碰撞检测逻辑写一个“无敌”外挂，只需要在进入这个流程的时候，把图 10–1 中用方框框起来的 jz 指令改成 jnz 指令即可。

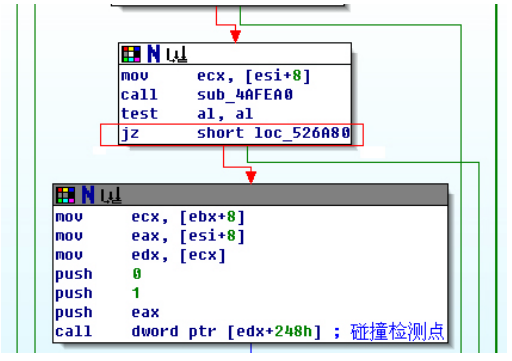


图 10-1　碰撞检测逻辑

那么，反外挂的逆向分析人员要怎样做才能快速定位被修改的 jz 指令地址呢？这里提供两种方法：一是使用现成的工具 Beyond Compare，二是自己写一个分析工具。但是，这两种思路的工作原理是一样的，那就是 dump 和 compare —— 先 dump 一份“干净”的游戏进程，再 dump 一份被外挂“感染”的游戏进程，然后将两者进行对比。不过，在分析外挂的过程中，大多数人还是喜欢和倾向于使用自己编写的工具，因为这样做可以提高工作效率。下面就简单地用分析工具演示一下分析修改代码外挂的思路。

将 Notepad.exe 作为游戏进程，分析工具是 GameSpider（第 8 章中有专门介绍）。当把 GameSpider 分析模块注入 Notepad.exe 进程后，需要执行 sam 命令来保存进程中所有的模块到指定的目录，如图 10–2 所示。通过 em 命令查看 notepad.exe 模块的信息，如图 10–3 所示。

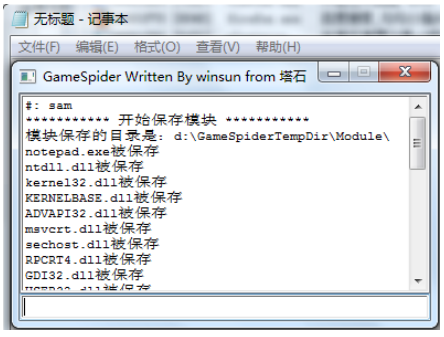


图 10-2　保存模块

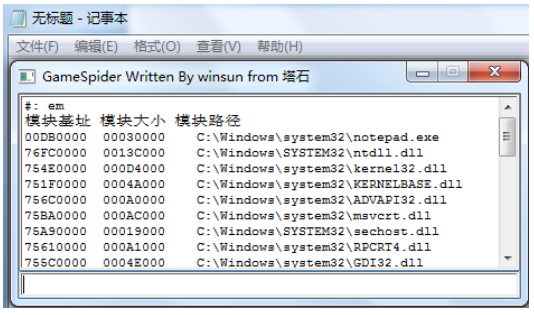


图 10-3　查看模块信息

地址 0xdb3689 处的代码如图 10–4 所示。这时，把地址 0xfd3689 处的代码改成 0x90，如图 10–5 所示。

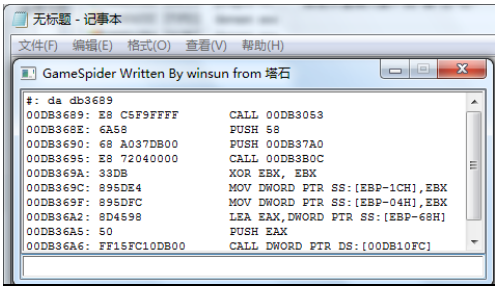


图 10-4　查看 0xdb3689 处的代码

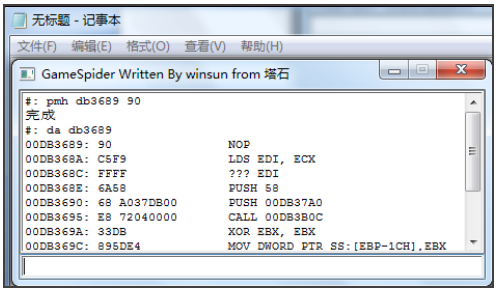


图 10-5　修改 0xdb3689 处的代码

用 cmf 命令来对内存中的 notepad.exe 映像和之前保存的 Notepad.exe 文件进行 .text 段的比较，如图 10–6 所示。

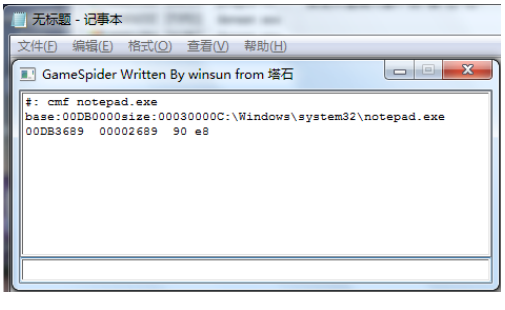


图 10-6　比较修改代码的情况

可以看到，代码在地址 0xdb3689 处被修改，所以，使用工具可以进行内存和文件的特定地址、特定长度的比较，其效果和效率都不比 Beyond Compare 差。

10.2　未授权调用检测

Call 函数是外挂最常用的一项功能，如自动攻击、施放技能等都会用到 Call 函数。本书第 5 章已经讨论过 Call 函数的用法，这里只简单讲解一下分析思路。因为目前的 Windows 客户端游戏大多采用 C++ 语言来编写，而 C++ 的多态机制更是在游戏编程中被广泛使用，所以，Call 函数的外挂中很大一部分是在 Call 虚函数。如果有办法对某个对象的所有虚函数实施监控，就能很快定位外挂 Call 的是哪个函数。否则，一个游戏中有几万个函数，我们怎么会知道外挂或游戏 Call 的是哪个函数呢？

对于反外挂的分析人员来说，监控虚函数调用只是第一步，第二步是对截获的虚函数调用进行 Call Stack 检测，这才是反外挂的关键。

程序的代码是由函数组成的，而程序执行的每个动作都有一个对应的执行流程，在这个流程上有相应的函数调用链。某款游戏的攻击动作所引发的执行流程或函数调用关系如图 10–7 所示。

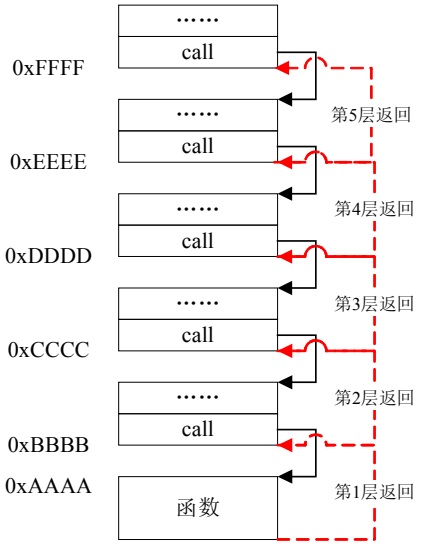


图 10-7　攻击调用和返回链

在图 10–7 中，实线表示函数调用链，虚线表示函数返回链。只要 Hook 地址 0xAAAA，就可以在整个调用链上进行 Call Stack 检测了。

同理，只要外挂作者分析出这条与攻击相关的调用链，然后在链上的某个节点 Call 某个函数，即可实现自动攻击功能。如图 10–8 所示，外挂 Call 游戏第 2 层所在的函数，结果原有的函数调用链就变成了现在这个样子。

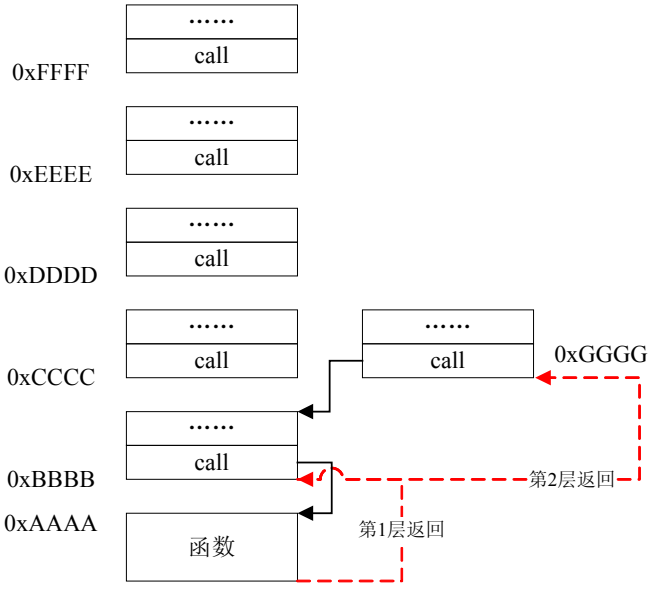


图 10-8　外挂未授权调用的堆栈回溯

在图 10–8 中，只要对地址 0xAAAA 处的函数进行多层堆栈回溯检测就会发现，从第 2 层开始，后续的返回地址与在没有使用外挂的情况下截然不同，这里再次体现了差异分析的思想。例如，正常情况下，第 2 层返回的地址是 0xBBBB，而使用外挂之后，第 2 层返回地址是 0xGGGG，前一个是在一个合法的模块地址空间，而后一个是在一个非法的地址空间，所以，通过这次对比，我们不仅能发现非法的调用，而且能够意外地获取外挂所在地址空间的信息。

从上面的分析来看，常规的 Call 函数还是比较容易检测出来的。但是，由于 Call 函数的功能强大，常规的 Call 又容易被检测到，所以，为了躲避检测，就出现了第 5 章中介绍的各种隐蔽 Call。

10.3　数据篡改检测

游戏里面有很多对象，如角色对象、怪物对象、物品对象、效果对象、精灵对象等。这些对象在某个时刻的状态是由对象的属性（即关键数据）来表示的，而读取和修改这些数据一般是通过对象本身的函数来实现的。对外挂有吸引力的关键数据有速度、HP、MP、物理攻击力、魔法攻击力、坐标等。本节将通过对一个吸怪挂的讲解来说明分析改数据挂的难度，最后引出线程转移和消息分流的设计机制。

10.3.1　吸怪挂分析

“吸怪”是指在短时间内将一定数量的怪物“吸”到角色周围，通常采用修改怪物坐标的方式实现。在游戏中，每个对象都有自己的坐标，角色和怪物也不例外。如果我们能读取角色的坐标，然后把角色当前的坐标赋值给怪物，那么怪物自然就会被吸附到角色周围了。

下面就让我们看看某款吸怪挂是如何实现吸怪的。为了简化对问题的描述，假设游戏中角色和怪物的坐标内存布局如图 10–9 所示。

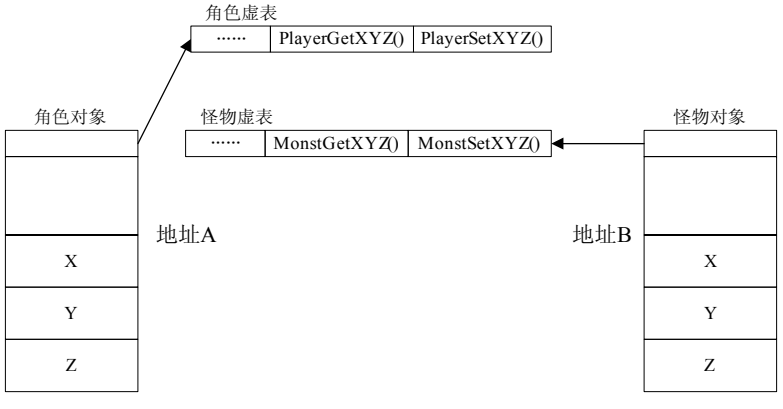


图 10-9　角色和怪物坐标内存布局

角色虚表中有两个函数 PlayerGetXYZ() 和 PlayerSetXYZ()，它们分别是角色获取自身当前坐标值函数 (X, Y, Z) 和设置自身当前坐标函数 (X, Y, Z)。在怪物虚表中，

MonstGetXYZ() 函数和 MonstSetXYZ() 函数则分别是怪物获取和设置自身当前坐标的函数。

通过对图 10–9 的分析，我们可以得到两种实现吸怪功能的方式：一种是调用 PlayerGetXYZ() 函数获取角色当前坐标值，然后把这个坐标值作为参数，调用 MonsterSetXYZ() 函数来设置怪物当前的坐标；另一种更为直接，读取地址 A 处的角色坐标值，然后直接赋值到地址 B 处怪物的坐标存放地址中。

对于反外挂分析人员来说，很明显，第一种吸怪的方式很容易被分析和防御，因为只要 Hook MonsterSetXYZ() 函数并进行 Call Stack 检测，就能马上检测出外挂 Call MonsterSetXYZ() 函数的操作。但是，对于第二种吸怪方式，分析和防御的难度就比较大了，因为存放角色和外挂对象坐标值的地址一般在堆数据区，直接读取和修改数据的操作很难被检测到，特别是在存放坐标值的地址动态变化的情况下，就更难分析了。当然，如果存放坐标的地址不是动态变化的，可以通过调试器下一个写断点，看看是谁在写这个地址。不过，游戏本身会不停调用 MonsterSetXYZ() 函数来修改自身的当前坐标，这样就会因为游戏自身的修改操作导致调试器中断游戏进程，进而干扰我们的分析。不过，也可以过滤游戏对 MonsterSetXYZ() 函数的调用，这样就不会影响我们的分析了。

上面讨论的吸怪方法，都是以知道坐标数据的内存布局，而且也知道是上面的函数在读取和修改这些数据为前提的。但是，如果我们第一次碰到吸怪挂，不知道吸怪是怎么回事，不清楚一般会用什么方法来实现吸怪，而且游戏是代理的，没有源代码，分析任务又迫在眉睫，该怎么办呢？有没有比较通用的方法来快速定位呢？第 10.3.2 节将要讨论的就是如何快速分析外挂是否修改了游戏中的未知数据以及如何快速定位被修改的数据。

10.3.2 线程转移和消息分流

线程转移和消息分流是为了解决定位外挂修改游戏数据的问题而设计的。我们知道，游戏都有一个主线程，这个主线程负责接收用户输入的消息，分发消息和游戏逻辑（AI、渲染、物理等）。最初的想法是，如果能把这个主线程 suspend，那么在外挂修改前后 dump 两份内存进行内存快照对比，不就能马上发现外挂修改数据的行

为了吗？但事与愿违，当我们把游戏主线程 suspend 之后，有一种情况是游戏反外挂系统会反对我们这样做；另一种情况是外挂无法输入任何消息，界面会被“冻住”，这是因为外挂也是基于主线程的消息分发机制来获取消息的，所以，suspend 主线程的方法就不可取了。那么，我们是不是可以向游戏主线程插入一个消息钩子，自己来分发消息，当是外挂的输入消息时候就放行，其余消息一律过滤（不分发）呢？结果也不尽人意。因为这种天真的想法忽略了游戏除了主线程的消息分发会引发一系列回调，主线程的游戏逻辑也会导致游戏数据本身的改变，所以，这种方式会造成大量的数据变化，无法看出是游戏本身修改的还是外挂修改的，也不可取。

“线程转移 + 消息分流”算是各种方案中比较可行的一种。这个方案会尽可能使游戏程序得不到执行，而使外挂和分析工具得到执行，以保持游戏代码和数据的静止状态。这种方法用来检查 Call 函数和改代码，效果极好。对于修改数据的外挂，虽然不能拿到满分，但还是有实际意义的。

下面就让我们开始真正的转移和分发之旅吧。根据《Windows 游戏编程大师》这本书中对游戏逻辑的分析，笔者画了一幅线程转移前后的对比图，如图 10–10 所示。

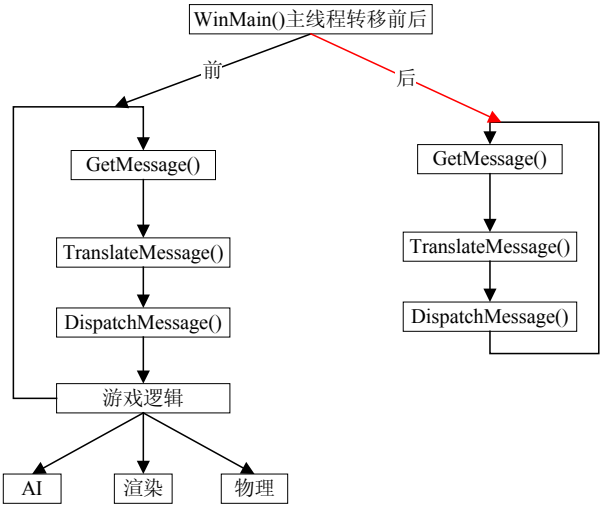


图 10-10　线程转移前后对比

可以看到，转移前游戏主线程就是在不停地执行 WinMain() 函数的一个循环，这个循环从 GetMessage() 函数开始，经过 TranslateMessage() 函数、DispatchMessage() 函

数，然后是游戏逻辑，最后进入 GetMessage() 函数，周而复始地执行消息队列取消息、分发消息和游戏逻辑的过程。在主线程转移后，WinMain() 函数只执行从消息队列取消息、转译消息和分发消息的操作，并不执行后面的游戏逻辑。这样一来，就相当于暂停了游戏逻辑，能“砍”掉很大一部分游戏本身修改数据的行为。

下面就让我们从代码的角度来看看如何实现线程转移（详细代码见相应章节）。

```
    // 转移线程
    BOOL TransferThread(DWORD dwDstThreadID, BYTE bOpenOrClose)
    {
       HANDLE hDstThread, hNewThread;
       DWORD dwNewThreadID;
       OutputDbgInfo(("[!] TransferThread Enter!\n"));
       __try
       {

    // 省略部分代码

           // 打开目标线程，获取目标线程句柄，全局保存
           g_hThreadHandle    =    OpenThread(THREAD_ALL_ACCESS,FALSE,
dwDstThreadID);
           if ( !g_hThreadHandle )
           {
               OutputDbgInfo(("[-] OpenThread fail! %s\n", FLINFO));
               return false;
           }

              // 创建功能线程，负责悬挂目标线程和转移目标线程
              hNewThread = CreateThread(NULL,
                           0,
     (LPTHREAD_START_ROUTINE)ThreadFuncForTransfer,
                           NULL,
                           0,
    &dwNewThreadID);
           if (!hNewThread)
```

```
            {
              OutputDbgInfo(("[-]  TransferThread  CreateThread  fail!\
n"));
                return FALSE;
            }
            CloseHandle(hNewThread);
            hNewThread = NULL;

    // 省略部分代码

        }
        __except(EXCEPTION_EXECUTE_HANDLER)
        {
            OutputDbgInfo(("[!] TransferThread Exception Exit!\n"));
            if ( hNewThread )
            {
                CloseHandle(hNewThread);
            }
            if (hDstThread)
            {
                CloseHandle(hDstThread);
            }
        }
        OutputDbgInfo(("[!] TransferThread Exit!\n"));
        return true;
    }
```

上面的 TransferThread() 函数打开了目标线程的句柄，创建了实现真正用于转移线程的 ThreadFuncForTransfer 线程。下面再让我们看看该函数的具体实现。

```
    // 实现真正的转移线程功能
    DWORD ThreadFuncForTransfer(LPVOID lpThreadParam)
    {
        HANDLE hDstThread;
        DWORD  dwMsgAddr;
```

```
        CONTEXT ctx = {0}, MidleCtx = {0};
        OutputDbgInfo(("[!] ThreadFuncForTransfer Enter!\n"));
        __try
        {

    // 省略部分代码

                // 悬挂目标线程
                SuspendThread(g_hThreadHandle);

                // 获取线程目标线程的 CONTEXT
                ctx.ContextFlags = CONTEXT_FULL;
                if (!GetThreadContext(g_hThreadHandle, &ctx))
                {
                    OutputDbgInfo(("[!] ThreadFuncForTransfer GetThread
Contextfail!\n"));
                    return FALSE;
                }

                // 保存原始 CONTEXT
                memcpy(&g_Ctx, &ctx, sizeof(CONTEXT));

                // 获取伪消息循环函数的地址
                dwMsgAddr = GetMsgLoopFuncAddr();

                // 重新设置目标线程的 EIP
                ctx.ContextFlags = CONTEXT_FULL;
                ctx.Eip = dwMsgAddr + 2;

                // 重新设置目标线程的 CONTEXT
                SetThreadContext(g_hThreadHandle, &ctx);

                // 恢复目标线程
                ResumeThread(g_hThreadHandle);
                Sleep(30);
```

```
        }
        __except(EXCEPTION_EXECUTE_HANDLER)
        {
            OutputDbgInfo(("[!]     ThreadFuncForTransfer     Exception
Exit!\n"));
        }
        OutputDbgInfo(("[!] ThreadFuncForTransfer Exit!\n"));
        return 1;
    }
```

上面的 ThreadFuncForTransfer() 函数实现了线程的转移，其中有一个获取伪消息循环的函数 GetMsgLoopFuncAddr()，它的具体实现如下。

```
    typedef DWORD(WINAPI* pFakeMsgLoop)(DWORD);
    pFakeMsgLoop g_pfMsgLoop = FakeMsgLoop;
    DWORD  __declspec(naked)  GetMsgLoopFuncAddr(void)
    {
        __asm
        {
            call $+5
           pop eax
            ret
            // 调用伪消息函数
            push 0
            call DWORD PTR [g_pfMsgLoop]
        }
    }
```

在上面的 GetMsgLoopFuncAddr() 函数中，执行“call $+5”指令后，首先把“pop eax”指令所在的地址压入栈内，然后跳转到“pop eax”指令去执行；执行完“pop eax”指令后，eax 的值就是“pop eax”指令所在的地址；最后执行“ret”指令，返回调用处“dwMsgAddr = GetMsgLoopFuncAddr()”，所以，在“ctx.Eip = dwMsgAddr + 2”指令中，ctx.Eip 指向上面“push 0”指令所在的地址，也就是说，主线程重新从“push 0”指令开始执行，当执行到“call DWORD PTR [g_pfMsgLoop]”指令后，就开始下面的伪消息循环。

```
// 伪消息循环函数
DWORD __stdcall FakeMsgLoop(DWORD)
{
   MSG msg;
   while(GetMessage(&msg, NULL, 0, 0))     // 从消息队列中取出所有消息
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }
   return 0;
}
```

经过上面的线程转移后，游戏主线程就只剩下执行一个消息循环了。但是，这样做还是不能杜绝主线程接收到消息后，通过 DispatchMessage() 函数引发回调来影响数据的修改，需要进一步对消息进行分流。分流的意思是让分发函数只分发分析工具和外挂的输入消息，对其他消息一律不分发。插入消息分流的逻辑图如图 10–11 所示，我们只要在之前的伪消息循环中插入消息分流规则，就可以实现消息分流了。

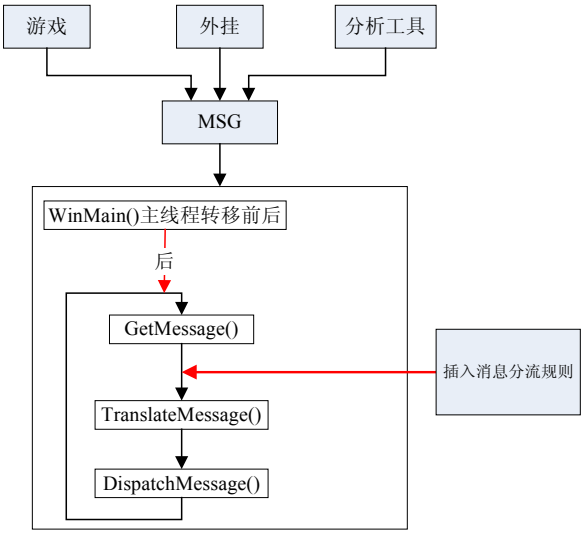


图 10-11　插入消息分流

下面就让我们看看插入消息分流规则后 FakeMsgLoop() 函数的实现吧。

```
// 伪消息循环函数
DWORD __stdcall FakeMsgLoop(DWORD)
{
   MSG msg;
   WORD wKey;
   while(GetMessage(&msg, NULL, 0, 0))  //从消息队列取所有消息
   {
      wKey = (WORD)msg.wParam;
      // 插入的消息分流规则
      if (msg.hwnd == g_hGameMainWnd && wKey != VK_F7)
      {
// 当消息来自游戏界面，且按键不是【F7】的时候继续消息循环
// 相当于过滤了所有来自游戏界面的非【F7】按键的消息
          continue;
      }

      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }
   return 0;
}
```

经过线程转移和消息分流之后，我们就可以置游戏于相对静止的理想状态了。

通过实践，在这种状态下，游戏本身造成的数据变化很少。但是，外挂的某些修改数据功能，一般只修改 1 字节或 4 字节，而经过转移和分流后，虽然游戏本身造成的数据修改很少，但相对于外挂修改的数据，还是偏多的。另外，游戏进程空间中的一些模块可能会回调游戏主线程所在模块的函数，游戏中的定时器等会有回调，而所有这些微弱的影响，也会修改一些数据。所以，线程转移和消息分流还是不能百分之百准确定位外挂所修改的数据的地址。不过，这种思路至少大大缩小了发生变化的地址范围。

10.4 本章小结

本章让读者站在“攻”与“防”中“防”的一面来思考问题——思考如何快速、可靠地检测外挂。虽然，代码篡改和未授权调用比较容易检测出来，但是数据篡改检测确实是一个比较难的课题，希望读者可以设计出更好的检测方案。

附录 A　声明

鉴于本书涉及的游戏安全技术具有破坏游戏公平性的风险，建议读者在学习、研究、探讨之前，确保已经充分了解以下内容：

本书所讨论的技术仅用于研究和学习，旨在提高游戏的安全性，严禁用于不良动机，任何个人、团队、组织不得将其用于非法目的，否则后果自负，特此声明。

附录 B　中国计算机安全相关法律及规定

（1）计算机信息系统的含义

1994 年 2 月 18 日，国务院发布的《中华人民共和国计算机信息系统安全保护条例》第 2 条做了如下规定：

本条例所称的计算机信息系统，是指由计算机及其相关的和配套的设备、设施（含网络）构成的，按照一定的应用目标和规则对信息进行采集、加工、存储、传输、检索等处理的人机系统。

（2）计算机病毒的含义

1994 年 2 月 18 日，国务院发布的《中华人民共和国计算机信息系统安全保护条例》第 28 条做了如下规定：

计算机病毒，是指编制或者在计算机程序中插入的破坏计算机功能或者毁坏数据，影响计算机使用，并能自我复制的一组计算机指令或者程序代码。

（3）非法侵入计算机信息系统罪

《中华人民共和国刑法》第二百八十五条　违反国家规定，侵入国家事务、国防建设、尖端科学技术领域的计算机系统的，处三年以下有期徒刑或者拘役。

（4）破坏计算机信息系统罪

《中华人民共和国刑法》第二百八十六条　违反国家规定，对计算机信息系统功能进行删除、修改、增加、干扰，造成计算机信息系统不能正常运行，后果严重的，处五年以下有期徒刑或者拘役；后果特别严重的，处五年以上有期徒刑。

违反国家规定，对计算机信息系统中存储、处理或者传输的数据和应用程序进行删除、修改、增加操作，后果严重的，依照前款的规定处罚。

故意制作、传播计算机病毒等破坏性程序，影响计算机系统，后果严重的，依照第一款的规定处罚。

（5）全国人民代表大会常务委员会《关于维护互联网安全的决定》（2000 年 12 月 28 日）（节选）

一、为了保障互联网的运行安全，对有下列行为之一，构成犯罪的，依照刑法有关规定追究刑事责任：

（一）侵入国家事务、国防建设、尖端科学技术领域的计算机信息系统；

（二）故意制作、传播计算机病毒等破坏性程序，攻击计算机系统及通信网络，致使计算机系统及通信网络遭受损害；

（三）违反国家规定，擅自中断计算机网络或者通信服务，造成计算机网络或者通信系统不能正常运行。

二、为了维护国家安全和社会稳定，对有下列行为之一，构成犯罪的，依照刑法有关规定追究刑事责任：

（一）利用互联网造谣、诽谤或者发表、传播其他有害信息，煽动颠覆国家政权、推翻社会主义制度，或者煽动分裂国家、破坏国家统一；

（二）通过互联网窃取、泄露国家秘密、情报或者军事秘密；

（三）利用互联网煽动民族仇恨、民族歧视，破坏民族团结；

（四）利用互联网组织邪教组织、联络邪教组织成员，破坏国家法律、行政法规实施。